

z/OS



Language Environment Programming Guide for 64-bit Virtual Addressing Mode

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 203.

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

This edition replaces SA22-7569-05.

© **Copyright IBM Corporation 2004, 2011.**

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

Using your documentation xi

How to read syntax diagrams xii

 Symbols xii

 Syntax items xii

 Syntax examples xiii

This Programming Guide xiv

Determining if a publication is current xv

Where to find more information xv

 Information updates on the web xv

 The z/OS Basic Skills Information Center xv

How to send your comments to IBM xvii

If you have a technical problem xvii

Summary of changes xix

Changes made in z/OS Version 1 Release 13 xix

Changes made in z/OS Version 1 Release 12 xix

Changes made in z/OS Version 1 Release 11 xix

Part 1. Creating AMODE 64 applications with Language Environment 1

Chapter 1. Introduction to Language Environment for AMODE 64 applications 3

Components of Language Environment for AMODE 64 applications. 3

Common runtime environment of Language Environment for AMODE 64 applications. 4

Chapter 2. Preparing to bind and run under Language Environment 7

Understanding the basics 7

 Planning to bind and run 7

 Binding AMODE 64 applications 8

Downward Compatibility Considerations. 8

Checking which runtime options are in effect 10

Chapter 3. Building and using AMODE 64 Dynamic Link Libraries (DLLs) 13

Support for DLLs 13

DLL concepts and terms 13

Loading a DLL 14

 Loading a DLL implicitly. 14

 Loading a DLL explicitly 15

Managing DLLs when running DLL applications. 20

 Loading DLLs 20

 Sharing DLLs. 21

 Freeing DLLs. 22

Creating a DLL or a DLL application 22

Building a DLL 22

 Writing DLL code 22

 Compiling your DLL code 25

 Binding your DLL code 26

Building a DLL application 27

Creating and using DLLs. 29

DLL restrictions 30

 Improving performance 31

Chapter 4. Binding, loading, and running under batch 33

Basic binding and running under batch 33

 Specifying runtime options in the EXEC

 statement 33

 Specifying Run-Time Options with the CEEOPTS

 DD card 34

Providing bind input 34

 Writing JCL for the bind process 34

 Binder control statements. 37

Bind options 38

Running an AMODE 64 application under batch 39

 Program library definition and search order 39

Specifying runtime options 40

Chapter 5. Binding and executing AMODE 64 programs using z/OS UNIX System Services 41

Basic binding and running C/C++ applications under z/OS UNIX 41

Invoking a shell from TSO/E 41

Using the z/OS UNIX c89 utility to bind and create AMODE 64 executable files 41

Running z/OS UNIX AMODE 64 application programs using z/OS XL C/C++ functions. 42

 z/OS UNIX application program environments 42

 Placing an MVS Application Executable Program

 in the File System 43

 Running an MVS Executable Program from a

 z/OS UNIX Shell 43

 Running POSIX-enabled Programs. 43

Chapter 6. Using runtime options 47

Understanding the basics. 47

Specifying runtime options 47

 Order of precedence 49

 Specifying suboptions in runtime options 50

 Specifying runtime options and program

 arguments. 50

 Using _CEE_RUNOPTS to specify runtime

 options. 50

 C and C++ compatibility considerations. 50

CEEEXOPT invocation syntax	51
Parmlib Syntax	54
CEEOPTS DD Syntax	55

Part 2. Preparing an application to run with Language Environment 57

Chapter 7. Using Language Environment parameter list formats 59

Understanding the basics	59
Argument lists and parameter lists	59
Passing arguments between routines	59

Chapter 8. Making Your Application Reentrant 63

Understanding the Basics	63
Making Your C/C++ Program Reentrant	63
Natural Reentrancy	63
Constructed Reentrancy	63
Generating a Reentrant Program Executable for C or C++	64
Installing a Reentrant Load Module	64

Part 3. Language Environment concepts, services, and models 65

Chapter 9. Initialization and termination under Language Environment. 67

Understanding the basics	67
Language Environment initialization	67
Language Environment termination	67
What causes termination	67
What happens during termination	68
Managing return codes in Language Environment	69
How the Language Environment enclave return code is calculated	69
Setting and altering user return codes	70
Termination behavior for unhandled conditions	71
Determining the abend code	71
Program interrupt abend and reason codes	71

Chapter 10. Program model. 73

Understanding the basics	73
Language Environment program model terminology	73
Process	74
Enclave	75
Thread	76
The full Language Environment program model	76
Mapping the POSIX program model to the Language Environment program model	77
Key POSIX program entities and Language Environment counterparts	77
Scope of POSIX semantics	78

Chapter 11. Stack and heap storage 81

Understanding the basics	81
------------------------------------	----

Stack storage overview	82
Tuning stack storage	83
Heap storage overview	84
Using heap pools to improve performance	85
Tuning heap storage	87
User-created heap storage	88
Alternative vendor heap manager	88

Chapter 12. Language Environment Condition Handling Introduction. 91

Understanding the Basics	91
The Stack Frame Model	92
Resume Cursor	93
What Is a Condition in Language Environment?	93
Steps in Condition Handling	93
Enablement Step	93
Condition Step	94
Termination Step and the TERMTHDACT Runtime Option	95
Invoking exception handlers	95
Responses to Conditions	96
Condition Handling Scenarios	96
Scenario 1: Simple Condition Handling	96
Scenario 2: Exception Handler Present for Divide-by-Zero	97

Chapter 13. Language Environment and HLL Condition Handling Interactions. 99

Understanding the Basics	99
C Condition Handling Semantics	99
Comparison of C-Language Environment Terminology	100
Controlling Condition Handling in C	100
C Condition Handling Actions	102
C Signal Representation of S/370 Exceptions	105
C++ Condition Handling Semantics	106
Language Environment and POSIX Signal Handling Interactions	106
Synchronous POSIX Signal and Language Environment Condition Handling Interactions	107

Chapter 14. Using Condition Tokens 111

Understanding the Basics	111
The Effect of Coding the fc Parameter	111
Testing a Condition Token for Success	112
Testing Condition Tokens for Equivalence	112
Testing Condition Tokens for Equality	113
Effects of Omitting the fc Parameter	113
Understanding the Structure of the Condition Token	113
Using Symbolic Feedback Codes	115
Locating Symbolic Feedback Codes for Conditions	115
Including Symbolic Feedback Code Files	115
Condition Tokens for C Signals under C and C++	116
q_data Structure for Abends	117
q_data Structure for Arithmetic Program Interruptions	118
q_data Structure for Square-Root Exception	120

Chapter 15. Using and Handling Messages	121
Understanding the Basics	121
Creating Messages	121
Creating a Message Source File	122
Using the CEEBLDTX Utility	125
Files Created by CEEBLDTX	127
Creating a Message Module Table	131
Assigning Values to Message Inserts	132
Interpreting Run-Time Messages	133
Specifying National Language	134
Run-Time Messages with POSIX	134
Handling Message Output	135
Using Multiple Message Handling APIs	136
Chapter 16. Using Date and Time Services	137
Chapter 17. National Language Support	139
Understanding the Basics	139
Setting the National Language	139
Setting the Locale	139
Chapter 18. Locale Callable Services	141
Chapter 19. General Callable Services	143
Understanding the Basics	143
__cdump()	143
Specifying a Target Directory for CEEDUMPs	143
__librel()	144
Chapter 20. Math Services	145
Part 4. Specialized Programming Tasks	147
Chapter 21. Assembler Considerations	149
Understanding the Basics	149
Compatibility Considerations	149
Register Conventions	149
Considerations for Coding or Running assembler Routines	150
GOFF	150
Asynchronous Interrupts	150
Condition Handling	150
Access to the Inbound Parameter String	151
CELQSTRT, CELQMAIN, CELQFMAN	151
Mode Considerations	151
Language Environment Library Routine Retention (LRR)	151

Assembler Macros	151
CELQPRLG Macro — Generate a Language Environment-Conforming AMODE 64 Prolog	152
CELQEPLG Macro — Terminate a Language Environment-Conforming AMODE 64 Routine	154
CEERCB Macro — Generate an RCB Mapping	154
CEPCB Macro — Generate an PCB Mapping	154
CEEEDB Macro — Generate an EDB Mapping	154
CEELAA Macro — Generate an LAA Mapping	155
CEELCA Macro — Generate an LCA Mapping	155
CEECAA Macro — Generate a CAA Mapping	155
CEEDSA Macro — Generate a DSA Mapping	155
CEEDIA Macro — Generate an DIA Mapping	156
CELQCALL Macro — Call a Language Environment-Conforming AMODE 64 Routine	156
CEEPDDA Macro — Define a Data Item in the Writeable Static Area (WSA)	158
CEEPLDA Macro — Returns the Address of a Data Item Defined by CEEPDDA	159

Chapter 22. Using preinitialization services with AMODE 64	161
Understanding the basics	161
Using preinitialization	162
Macros that generate the PreInit table	162
Invoking CELQPIPI	164
CELQPIPI interface	165
Service Routines	179
An example program invocation of CELQPIPI	186

Part 5. Appendixes 197

Appendix. Accessibility	199
Using assistive technologies	199
Keyboard navigation of the user interface	199
z/OS information	199
Dotted decimal syntax diagrams	199

Notices	203
Policy for unsupported hardware	205
Permission Notice	205
Programming Interface Information	205
Trademarks	206

Bibliography	207
Language Products Publications	207
Related Publications	208

Index	211
--------------	------------

Figures

1. Components of Language Environment	4	19. Overview of the Full Language Environment Program Model	77
2. Language Environment's Common Runtime Environment	5	20. Scope of Semantics against POSIX Processes and Language Environment Processes/Enclaves	79
3. Explicit use of a DLL in an application using the dllload() family of functions.	16	21. Language Environment Stack Storage Model	83
4. Explicit Use of a DLL in a application using the dlopen() family of functions.	18	22. Language Environment Heap Storage Model	85
5. Using #pragma export to Create a DLL Executable Module Named BASICIO	22	23. Condition Processing	94
6. Using #pragma Export to Create a DLL Executable Module TRIANGLE	23	24. Scenario 1: Division by Zero with No User Exception Handlers Present	96
7. Using _export to Create DLL Executable Module TRIANGLE.	23	25. Scenario 2: Division by Zero with a User Handler Present in Routine B	97
8. Using Language Environment macros to create an AMODE 64 assembler DLL executable named ADLL6EV2	25	26. C370A Routine	103
9. Assembler DLL Application calling an assembler DLL	29	27. C370B Routine	103
10. Summary of DLL and DLL Application Preparation and Usage.	30	28. C370C Routine	104
11. Basic Batch bind Processing	34	29. C Condition Handling Example	104
12. Creating an AMODE 64 Executable Program under Batch	37	30. Enablement Step for Signals under z/OS UNIX	108
13. Using the INCLUDE binder Control Statement	37	31. Language Environment Condition Token	114
14. Using the LIBRARY binder Control Statement	38	32. Structure of Abend Qualifying Data	118
15. Sample invocation of CEEXOPT within CELQUOPT source program.	52	33. q_data Structure for Arithmetic Program Interruption Conditions	119
16. Call Terminology Refresher	59	34. Example of a Message Source File	122
17. Argument Passing Styles in Language Environment	60	35. Example of a Message Module Table with One Language	131
18. Program model illustration of resource ownership	74	36. Example of a Message Module Table with Two Languages	132
		37. Service routines.	179
		38. 64-bit function descriptors	180
		39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment	186
		40. AMODE 64 C Main PreInit Routine	195

Tables

1. How to Use z/OS Language Environment Publications	xi	17. Mapping of Abend Signals to C Signals	106
2. Syntax examples	xiii	18. Language Environment Condition Tokens and Non-POSIX C Signals	116
3. Prerequisite z/OS Release Level for Compilers that Support Downward Compatibility	10	19. Language Environment Condition Tokens and POSIX C Signals	116
4. Required Data Sets Used for binding	35	20. Arithmetic Program Interruptions and Corresponding Conditions	118
5. Optional Data Sets Used for binding	36	21. Square-Root Exception and Corresponding Condition	120
6. Selected bind options	38	22. Language Environment Run-Time Message Severity Codes	133
7. Formats for specifying runtime options and program arguments	50	23. Condition Tokens with POSIX	134
8. Semantic Terms and Methods for Passing Arguments in Language Environment.	60	24. C/C++ Redirected Stream Output	136
9. Default Passing Style per HLL	61	25. Preinitialization Services Accessed Using CELQPIPI.	165
10. Summary of Enclave Reason Codes	70	26. program_attributes mask values	178
11. Abend Code Values Used by Language Environment	71	27. Return and reason codes.	181
12. Program Interrupt Abend and Reason Codes	71	28. Return and reason codes.	182
13. Usage of Stack and Heap Storage by Language Environment-Conforming Languages	81	29. PreInit Storage Attributes Control Block Field Descriptions	182
14. Language Environment Default Responses to Unhandled Conditions	95	30. Return and reason codes.	184
15. C Conditions and Default System Actions	100	31. Return and reason codes.	185
16. Mapping of S/370 Exceptions to C Signals	105	32. Return and reason codes.	186

About this document

This document supports z/OS (5694–A01).

IBM® z/OS Language Environment (also called Language Environment) for AMODE 64 application provides common services and language-specific routines in a single runtime environment for C, C++, and assembler applications. It offers consistent and predictable results for language applications, independent of the language in which they are written.

Language Environment is the prerequisite runtime environment for applications generated with z/OS XL C/C++.

Language Environment consists of the common execution library (CEL) and the runtime libraries for C/C++.

Using your documentation

The publications provided with Language Environment are designed to help you:

- Manage the run-time environment for applications generated with a Language Environment-conforming compiler.
- Write applications that use the Language Environment callable services.
- Develop interlanguage communication applications.
- Customize Language Environment.
- Debug problems in applications that run with Language Environment.
- Migrate your high-level language applications to Language Environment.

Language programming information is provided in the supported high-level language programming manuals, which provide language definition, library function syntax and semantics, and programming guidance information.

Each publication helps you perform different tasks, some of which are listed in Table 1. All books are available in printable (PDF) and BookManager softcopy formats. For a complete list of publications that you may need, see “Bibliography” on page 207.

Table 1. How to Use z/OS Language Environment Publications

To ...	Use ...
Evaluate Language Environment	<i>z/OS Language Environment Concepts Guide</i>
Plan for Language Environment	<i>z/OS Language Environment Concepts Guide</i> <i>z/OS Language Environment Run-Time Application Migration Guide</i>
Install Language Environment	<i>z/OS Program Directory</i>
Customize Language Environment	<i>z/OS Language Environment Customization</i>

Table 1. How to Use z/OS Language Environment Publications (continued)

To ...	Use ...
Understand Language Environment program models and concepts	<i>z/OS Language Environment Concepts Guide</i>
	<i>z/OS Language Environment Programming Guide</i>
	<i>z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode</i>
Find syntax for Language Environment run-time options and callable services	<i>z/OS Language Environment Programming Reference</i>
Develop applications that run with Language Environment	<i>z/OS Language Environment Programming Guide</i> and your language programming guide
Debug applications that run with Language Environment, diagnose problems with Language Environment	<i>z/OS Language Environment Debugging Guide</i>
Get details on run-time messages	<i>z/OS Language Environment Run-Time Messages</i>
Develop interlanguage communication (ILC) applications	<i>z/OS Language Environment Writing Interlanguage Communication Applications</i> and your language programming guide
Migrate applications to Language Environment	<i>z/OS Language Environment Run-Time Application Migration Guide</i> and the migration guide for each Language Environment-enabled language

How to read syntax diagrams

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

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

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

Symbols

The following symbols may be displayed in syntax diagrams:

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

Syntax items

Syntax diagrams contain many different items. Syntax items include:

- Keywords - a command name or any other literal information.
- Variables - variables are italicized, appear in lowercase, and represent the name of values you can supply.

- Delimiters - delimiters indicate the start or end of keywords, variables, or operators. For example, a left parenthesis is a delimiter.
- Operators - operators include add (+), subtract (-), multiply (*), divide (/), equal (=), and other mathematical operations that may need to be performed.
- Fragment references - a part of a syntax diagram, separated from the diagram to show greater detail.
- Separators - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.

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

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

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

Syntax examples

The following table provides syntax examples.

Table 2. Syntax examples

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

Table 2. Syntax examples (continued)

Item	Syntax example
Default.	
Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.	
Variable.	
Variables appear in lowercase italics. They represent names or values.	
Repeatable item.	
An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.	
A character within the arrow means you must separate repeated items with that character.	
An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.	
Fragment.	
The fragment symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.	

This Programming Guide

To use this book you should be familiar with the Language Environment product and C/C++ or assembler. C/C++ is used generically to refer to information that applies to both C and C++.

For application programming, you will need to use this book and *z/OS Language Environment Programming Reference*. This book contains information about binding, running, and using services within the Language Environment environment, the Language Environment program model, and language- and operating system-specific information, where applicable. *z/OS Language Environment Programming Reference* contains more detailed information, as well as specific syntax for using runtime options and callable services.

This book is organized as follows:

- Part 1 includes a basic introduction to Language Environment. It also explains the steps for creating and running executable programs, and provides an overview of runtime options.
- Part 2 describes how to prepare an application to run in Language Environment.

- Part 3 describes Language Environment concepts, services, and models, including initialization and termination, program model, storage, condition handling, messages, and callable services.
- Part 4 addresses specialized programming tasks, such as assembler considerations, and preinitialization services.

Determining if a publication is current

As needed, IBM changes its information. Here's how to determine if you are looking at the most current copy of a book:

1. At the end of the order number there is a dash followed by two digits, often called the dash level. A book with a higher dash level is more current than one with a lower dash level. For example, in the book order number SC28-1939-01, the dash level 01 means that the book is more current than previous levels, such as 00.
2. If a hardcopy book and a softcopy book have the same dash level, it is possible that the softcopy book is more current than the hardcopy book. Check the dates shown in the Summary of Changes. The softcopy book might have a more recently dated Summary of Changes than the hardcopy book.
3. To compare softcopy books, you can check the last two characters of the softcopy filename (also called the book name). The higher the number, the more recent the book. For example, IEA4E802 is more recent than IEA4E801. Also, an asterisk next to the book titles in the CD-ROM booklet and the readme files indicates that a publication is either new or has been changed.

Where to find more information

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

Information updates on the web

For the latest information updates that have been provided in PTF cover letters and documentation APARs for z/OS[®], see the online document at:
http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/Shelves/ZDOCAPAR

This document is updated weekly and lists documentation changes before they are incorporated into z/OS publications.

The z/OS Basic Skills Information Center

The z/OS Basic Skills Information Center is a Web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. The Information Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, the z/OS Basic Skills Information Center is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS.

To access the z/OS Basic Skills Information Center, open your Web browser to the following Web site, which is available to all users (no login required):
<http://publib.boulder.ibm.com/infocenter/zos/basics/index.jsp>

How to send your comments to IBM

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

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

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

Include the following information:

- Your name and address
- Your email address
- Your telephone or fax number
- The publication title and order number:
z/OS V1R13.0 Language Environment Programming Guide for 64-bit Virtual Addressing Mode
SA22-7569-06
- The topic and page number related to your comment
- The text of your comment.

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

IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you submit.

If you have a technical problem

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

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

Summary of changes

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

Changes made in z/OS Version 1 Release 13

This document contains information that was previously presented in *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*, SA22-7569-05, which supports z/OS Version 1 Release 12.

Changed information:

- Minor editorial and formatting changes were made.

Changes made in z/OS Version 1 Release 12

This document contains information that was previously presented in *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*, SA22-7569-04, which supports z/OS Version 1 Release 11.

New information:

- Information about how to control overrideable and nonoverrideable run-time options through CEEPRMxx members has been added. For more information, see "Specifying runtime options" on page 47.

Changed information:

- Information about specifying run-time options is updated in Chapter 4, "Binding, loading, and running under batch," on page 33.
- Information about parmlib syntax is updated in Chapter 6, "Using runtime options," on page 47.
- The "Readers' Comments - We'd Like to Hear from You" section at the back of this publication has been replaced with a new section "How to send your comments to IBM" on page xvii. The hardcopy mail-in form has been replaced with a page that provides information appropriate for submitting readers comments to IBM.

Changes made in z/OS Version 1 Release 11

This document contains information that was previously presented in *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*, SA22-7569-03, which supports z/OS Version 1 Release 10.

New information:

- Information is added on invoking CELQPIPI. See "Invoking CELQPIPI" on page 164.
- Support for @GETSTORE, @FREESTORE, and @MSGRTN is added. See "Service Routines" on page 179.

Changed information:

- SIGTRACE information is added in

- "POSIX Signals that Do Not Enter Condition Handling" on page 109,
 - "Condition Tokens for C Signals under C and C++" on page 116, and
 - "Run-Time Messages with POSIX" on page 134.
- Information about `__moservices()` is added. See Chapter 11, "Stack and heap storage," on page 81.

Part 1. Creating AMODE 64 applications with Language Environment

This section explains the steps for creating and running an AMODE 64 executable program, and provides an overview of runtime options.

Note: The terms in this section having to do with linking (*bind*, *binding*, and so forth) refer to the process of creating an AMODE 64 executable program from object modules (the output produced by compilers and assemblers). The program used is the DFSMS program management binder. The binder extends the services of the linkage editor and is the default program provided for creating an executable.

Chapter 1. Introduction to Language Environment for AMODE 64 applications

Language Environment supports 64-bit addressing for applications written in C, C++, or *Language Environment-conforming* Assembler.

In the 64-bit addressing mode (AMODE 64) Language Environment supports addresses that are 64 bits in length, which allows access to data in virtual storage up to 16 exabytes. Hence applications that work with large databases or large volumes of data can consolidate data in one address space. There are a few things to note:

- An AMODE 64 Language Environment application supports XPLINK linkage only and the C runtime environment is always initialized. A new library, CELQLIB, is shipped for AMODE 64 support, while the existing libraries continue to be shipped and supported.
- Pre initialized environments are supported via CELQPIPI.
- The user stack and heap, along with most of Language Environment storage is above the 2GB bar.
- There is a new anchor for AMODE 64 applications so register 12 no longer needs to be reserved for the address of the CAA.
- The only means of communication between AMODE 64 and AMODE 24 or AMODE 31 applications is through mechanisms that can communicate across processes or address spaces. However, Language Environment applications that use AMODE 64 can run with existing applications that use AMODE 24 or AMODE 31 on the same physical zSeries system.
- Where necessary, there are new Language Environment runtime options to support AMODE 64 applications. The new runtime options primarily support the new stack and heap storage located above the bar. Some of the existing options are no longer available.

Components of Language Environment for AMODE 64 applications

As Figure 1 on page 4 shows, Language Environment for 64-bit Virtual Address Mode consists of the following components:

- Basic routines that support starting and stopping programs, allocating and managing storage, and indicating and handling error conditions, and providing debugging facilities.
- C/C++ runtime library services, including math services and date and time services, that are commonly needed by programs running on the system.

Language Environment

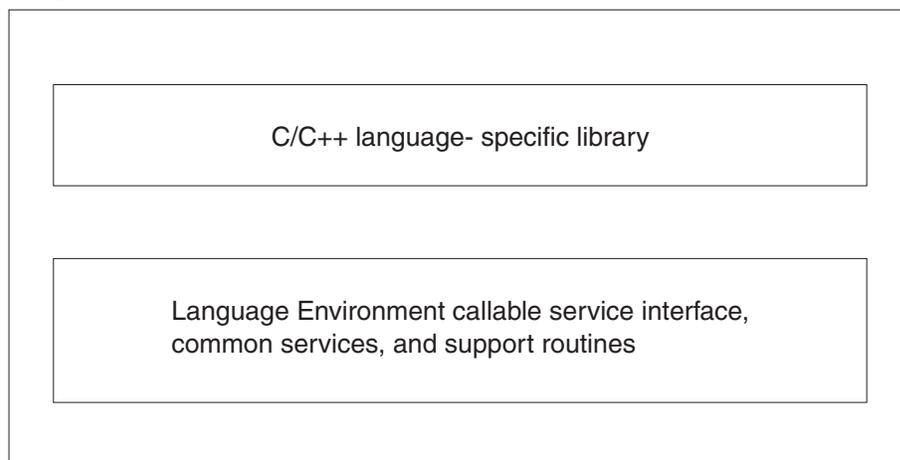


Figure 1. Components of Language Environment

The z/OS XL C/C++ compiler with the LP64 compiler option is the IBM Language Environment-conforming language compiler that currently supports 64-bit addressing.

Along with the change in addressing mode to use 64 bits, the other important consideration is that long data types also use 64 bits. The industry standard name for this data model is LP64, which translates roughly to "long and pointer data types use 64 bits." The Language Environment support for AMODE 24 and AMODE 31 applications is ILP32, meaning "integer, long, and pointer data types use 32 bits."

Language Environment also provides new Assembler macros that support creating Language Environment conforming Assembler applications that run AMODE 64.

Common runtime environment of Language Environment for AMODE 64 applications

Figure 2 on page 5 illustrates the common environment that Language Environment creates.

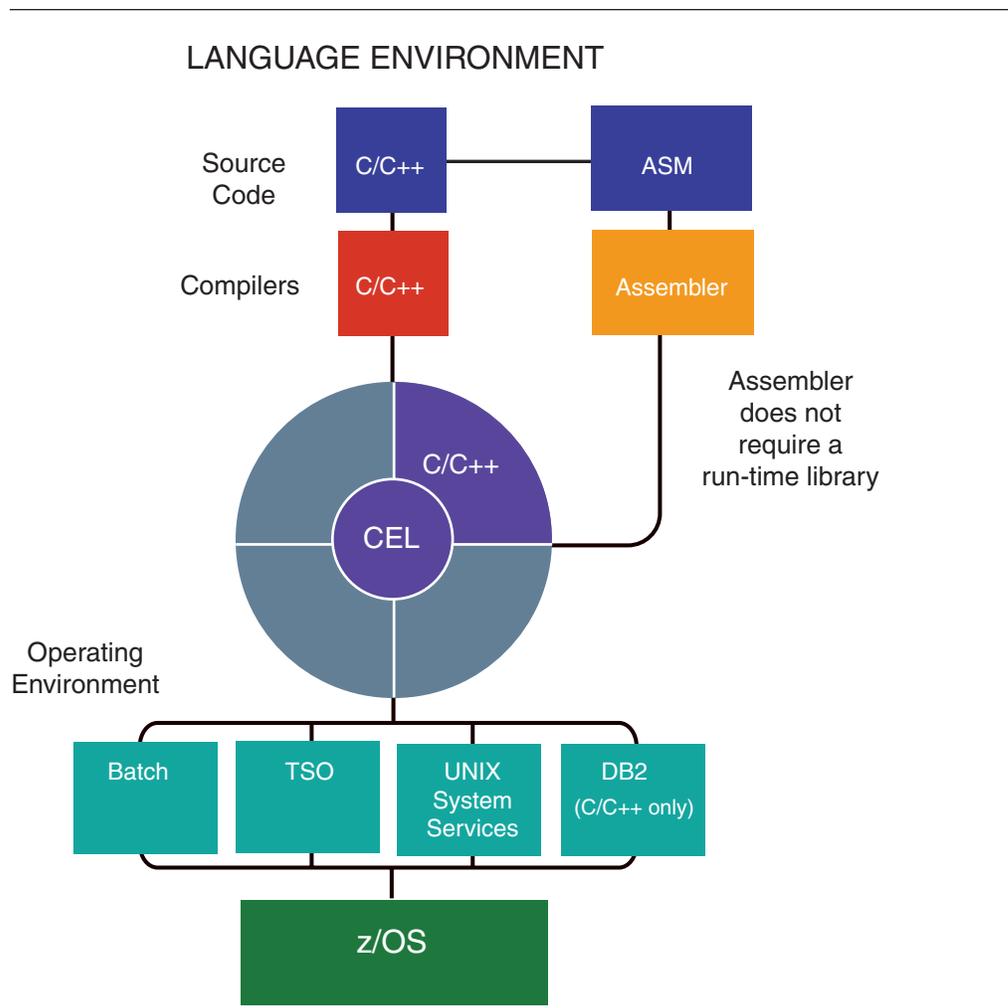


Figure 2. Language Environment's Common Runtime Environment

Chapter 2. Preparing to bind and run under Language Environment

This chapter describes the basic information that you need to know before binding and running AMODE 64 applications under Language Environment. The procedures are similar to that used by AMODE 24 and AMODE 31 applications.

This chapter is not intended to illustrate every aspect of binding and running you might want to learn. Detailed instructions about binding and running AMODE 64 applications are provided in subsequent chapters, and in *z/OS MVS Program Management: User's Guide and Reference*.

Note: The term "bind" is used through out this document to describe the process of converting compiler output into AMODE 64 executable programs. The "binder" is the name of the z/OS program that performs this process.

Understanding the basics

Language Environment library routines are divided into two categories: *resident routines* and *dynamic routines*. The resident routines are linked with the application and include such things as the bootstrap routines. The dynamic routines are not part of the application and are dynamically loaded during run time.

The way Language Environment code is packaged keeps the size of AMODE 64 application executable programs small. In most cases when you apply Language Environment maintenance, you need not re-bind the application code except under rare and special circumstances.

The binder converts an object module into an AMODE 64 executable program and stores it in a library. The executable program can then be run from that library at any time. The bind process combines output from compilers, language translators, bind programs and control statements to produce an executable program (program object) and stores it in a library (PDSE or HFS). The executable program can then be run from that library. Only the program management binder can be used to perform the bind process for AMODE 64 applications. See *z/OS MVS Program Management: User's Guide and Reference* for a complete discussion of services to create, load, modify, list, read, transport and copy AMODE 64 executable programs.

Planning to bind and run

There are certain considerations that you must be aware of before binding and running AMODE 64 applications under Language Environment.

Language Environment resident routines for AMODE 64 applications are located in the SCEEBND2 library. Language Environment dynamic routines are located in the SCEERUN and SCEERUN2 libraries. The Language Environment libraries are located in data sets identified with a high-level qualifier specific to the installation.

The following is a summary of the Language Environment libraries and their contents:

Preparing to Use Language Environment

SCEERUN

There are some members in this PDS, such as message catalogs, that are used by AMODE 64 applications.

SCEERUN2

A PDSE which contains the run-time library routines needed during execution of AMODE 64 applications.

SCEEBND2

Contains all Language Environment resident routines for AMODE 64 applications. It provides only a small number of resident routines, since most of the functions formerly provided in those static libraries are instead provided using dynamic linkage.

SCEELIB

Contains side-decks for DLLs provided by Language Environment.

Many of the APIs available to AMODE 64 applications appear externally as DLL functions. See Chapter 3, “Building and using AMODE 64 Dynamic Link Libraries (DLLs),” on page 13 for information about DLLs. To resolve these references from AMODE 64 applications, a definition side-deck must be included when binding the application. The SCEELIB library contains the following side-decks:

- CELQS003 — Side-deck to resolve references to callable services in the C/C++ run-time library when binding an AMODE 64 application.
- CELQSCPP — Side-deck to resolve references to C++ run-time library (RTL) definitions that may be required when binding an AMODE 64 application.

The functions in these side-decks can be called from an AMODE 64 application. However, they cannot be used as the target of an explicit `dllqueryfn()` or `dlsym()` against the DLL.

Binding AMODE 64 applications

The main entry point for an AMODE 64 application is CELQSTRT.

The XL C/C++ compiler generates CELQSTRT for `main()` and fetchable functions, and references to CELQSTRT for other functions. Language Environment-conforming AMODE 64 assembler programs do not generate CELQSTRT but do generate references to CELQSTRT.

A copy of CELQSTRT resides in the Language Environment SCEEBND2 library. Do not explicitly include it in the bind process. Instead avoid using the NCAL binder option and let the automatic library call mechanism of the binder resolve references to CELQSTRT.

Downward Compatibility Considerations

Language Environment-conforming AMODE 64 applications cannot run on any release prior to z/OS Version 1 Release 6. As of z/OS Version 1 Release 7, Language Environment provides downward compatibility support for Language Environment-conforming AMODE 64 applications. Assuming that required programming guidelines and restrictions (see below) are observed, this support enables programmers to develop applications on higher release levels of the operating system, for deployment on execution platforms that are running lower release levels of the operating system. For example, you may use z/OS V1R7 (and Language Environment) on a development system where applications are coded,

link edited, and tested, while using z/OS V1R6 (and Language Environment) on their production systems where the finished application modules are deployed.

Downward compatibility support is not the roll-back of new function to prior releases of the operating system. Applications developed exploiting the downward compatibility support must not use Language Environment function that is unavailable on the lower release of the operating system where the application will be deployed. The downward compatibility support includes toleration PTFs for lower releases of the operating system (specific PTF numbers can be found in the PSP buckets), to assist in diagnosis of applications that violate the programming requirements for this support.

The downward compatibility support provided by z/OS V1R7 and later, and by the toleration PTFs, does not change Language Environment's upward compatibility. That is, applications coded and link-edited with one release of Language Environment will continue to execute on later releases of Language Environment, without a need to recompile or relink-edit the application, independent of the downward compatibility support.

The application requirements and programming guidelines for downward compatibility are:

- The application must only use Language Environment function that is available on the release level of the operating system used on the target deployment system.
- The application must only use Language Environment function that is available on the release level of the operating system used for developing and link-editing the application, by using the appropriate Language Environment object libraries, header files, and macros.
- The release level of the operating system used for application development and link-editing must be at least the level that is the prerequisite of the compiler product(s) (XL C/C++) that are used to develop the application.
- The release level of the operating system used on the target deployment system must be at least the level that is the prerequisite of the compiler product(s) that are used to develop the application.
- The release level of the operating system used for application development and link-editing must be at least z/OS V1R6 for Language Environment-conforming AMODE 64 applications.
- The program object format of the application must be no greater than the highest level supported on the target deployment system.

The term "Language Environment function" used in the discussion of downward compatibility support refers to:

- Language Environment callable services (see *z/OS Language Environment Programming Reference*).
- Language Environment runtime options
- C/C++ library functions
- UNIX branding functions
- Other new language functionality that has an explicit operating system release prerequisite that is documented in the user publications. For example, with z/OS V1R6 Language Environment, new support was added for Language Environment-conforming AMODE 64 applications. This support is available on z/OS V1R6 Language Environment or later, but is not available on prior releases.

Preparing to Use Language Environment

The compiler products that support development of downward compatible applications are listed in the table below, along with their prerequisite minimum release level of the operating system. (Note that prior releases of the compilers beyond those listed in the table are still supported by Language Environment, but do not provide downward compatibility for Language Environment-conforming AMODE 64 applications. They only support *upward* compatibility.)

Table 3. Prerequisite z/OS Release Level for Compilers that Support Downward Compatibility

Compiler Product	z/OS release level prerequisite
z/OS XL C/C++ compiler	z/OS V1R6

The diagnosis assistance that will be provided by the toleration PTFs includes:

- **Options Processing:** Whenever an application exploits Language Environment run-time options that are unavailable on the release of the operating system the application is executed on, a message will be issued. In order to issue this message, toleration PTFs are available down to OS/390® V2R6, and you must apply them on the target system. The use of environment variables, even specific Language Environment ones, is not covered by this support.
- **Detection of unsupported function:** In many cases where a programmer disregards the requirements and programming guidelines and exploits a Language Environment function that is unavailable on the release of the operating system the application is executed on, Language Environment will raise a new condition. With an unhandled condition, the application is terminated. In order to raise this new condition, toleration PTFs are available down to OS/390 V2R6, and you must apply them on the target system.
- **C/C++ headers:** As of OS/390 V2R10, support has been added to the C/C++ headers shipped with Language Environment to allow application developers to "target" a specific release, in order to ensure the application hasn't taken advantage of any new C/C++ library function. See *z/OS XL C/C++ User's Guide* for details of how the TARGET compiler option can be used to create downward-compatible applications and prevent application developers from using new C/C++ library functions in applications.
- **Detection of unsupported program object format:** If the program object format is at a level which is not supported by the target deployment system, then the deployment system will produce an abend when trying to load the application program. The abend will indicate that DFSMS/MVS was unable to find or load the application program. Correcting this problem does not require the installation of any toleration PTFs. Rather the application developer will need to recreate the program object which is compatible with older deployment system. For information on using the Program Management binder COMPAT option, see *z/OS MVS Program Management: User's Guide and Reference*.

Checking which runtime options are in effect

Using the Language Environment runtime option RPTOPTS, you can control whether an Options Report is produced after a successful running of the application. The Options Report shows all of the runtime options that were in effect when the application began to run. The IBM-supplied default for RPTOPTS is OFF, meaning that the Options Report is not produced. If you override the default setting of RPTOPTS, using one of the mechanisms described in Chapter 6, "Using runtime options," on page 47, and the application completes successfully, the Options Report is written to the C stderr stream. For more information on the C stderr stream, see *z/OS XL C/C++ Programming Guide*.

For the syntax of RPTOPTS, see *z/OS Language Environment Programming Reference*.

Chapter 3. Building and using AMODE 64 Dynamic Link Libraries (DLLs)

The z/OS dynamic link library (DLL) facility provides a mechanism for packaging programs and data into program objects (DLLs) that may be accessed from other separate program objects. A DLL can *export* symbols representing routines that may be called from outside the DLL, and can *import* symbols representing routines or data or both in other DLLs, avoiding the need to link the target routines into the same program objects as the referencing routine. When an application references a separate DLL for the first time, it is automatically loaded into memory by the system.

An application that has been compiled and linked to be AMODE 64 uses the XPLINK linkage convention, which is automatically enabled for DLL support. There is no distinction between “DLL code” and “non-DLL code” as there is for AMODE 31 applications.

This chapter defines concepts and shows how to build DLLs and DLL applications.

Support for DLLs

DLL support is available for AMODE 64 applications running under the following systems:

- z/OS Batch
- TSO
- z/OS UNIX System Services

AMODE 64 applications compiled with the XL C and XL C++ compilers can create and reference DLLs. The High Level Assembler (HLASM) Release 5 can also be used to create AMODE 64 assembler routines if the supporting Language Environment macros are used. For more details on Language Environment assembler macro support, see Chapter 21, “Assembler Considerations,” on page 149.

DLL concepts and terms

Function

In this chapter, function is used to generically refer to a callable routine or program, and is specifically applicable to C and C++.

Variable

In this chapter, variable is used to generically refer to a data item, such as a static variable in C/C++.

Application

All the code executed from the time an AMODE 64 executable program is invoked until that program, and any programs it directly or indirectly calls, is terminated.

DLL An executable module that exports functions, variable definitions, or both, to other DLLs or DLL applications. The executable code and data are bound to the program at run time. The code and data in a DLL can be shared by several DLL applications simultaneously.

Dynamic Link Libraries (DLLs)

DLL application

An application that references imported functions, imported variables, or both, from other DLLs.

Executable program (or executable module)

A file which can be loaded and executed on the computer. For AMODE 64 applications, z/OS only supports program objects created by the binder that reside in either a PDSE or in the HFS.

Object code (or object module)

A file output from a compiler after processing a source code module, which can subsequently be used to build an AMODE 64 executable program module.

Source code (or source module)

A file containing a program written in a programming language.

Imported functions and variables

Functions and variables that are not defined in the executable module where the reference is made, but are defined in a referenced DLL.

Non-imported functions and variables

Functions and variables that are defined in the same executable module where a reference to them is made.

Exported functions or variables

Functions or variables that are defined in one executable module and can be referenced from another executable module. When an exported function or variable is referenced within the executable module that defines it, the exported function or variable is also nonimported.

Writable Static Area (WSA)

An area of memory that is modifiable during program execution. Typically, this area contains global variables and function and variable descriptors for DLLs. The environment supplied to an XPLINK function (in register 5) is the part of WSA that is applicable to that function.

Function descriptor

An internal control block containing information needed by compiled code to call a function.

Loading a DLL

A DLL is loaded implicitly when an application references an imported variable or calls an imported function. DLLs can be explicitly loaded by calling `dllload()` or `dlopen()`. Due to optimizations performed, the DLL implicit load point may be moved and is only done before the actual reference occurs.

Loading a DLL implicitly

When an application uses functions or variables defined in a DLL, the compiled code loads the DLL. This implicit load is transparent to the application. The load establishes the required references to functions and variables in the DLL by updating the control information contained in function descriptors and variable pointers.

If a C++ DLL contains static classes, their constructors are run when the DLL is loaded. Their destructors run once after the main function returns.

To implicitly load a DLL from C or C++, do one of the following:

- Statically initialize a variable pointer to the address of an exported DLL variable.
- Reference a function pointer that points to an exported function.
- Call an exported function.
- Reference (use, modify, or take the address of) an exported variable.
- Call through a function pointer that points to an exported function.

When the first reference to a DLL is from static initialization of a C or C++ variable pointer, the DLL is loaded before the main function is invoked. Any C++ constructors are run before the main function is invoked.

Loading a DLL explicitly

The use of DLLs can also be explicitly controlled by C/C++ application code at the source level. The application uses explicit source-level calls to one or more runtime services to connect the reference to the definition. The connections for the reference and the definition are made at run time.

The DLL application writer can explicitly call the following C runtime services:

- `dllload()`, which loads the DLL and returns a handle to be used in future references to this DLL
- `dllqueryfn()`, which obtains a pointer to a DLL function
- `dllqueryvar()`, which obtains a pointer to a DLL variable
- `dllfree()`, which frees a DLL loaded with `dllload()`

The following runtime services are also available as part of the Single UNIX Specification, Version 3:

- `dlopen()`, which loads the DLL and returns a handle to be used in future references to this DLL.
- `dldclose()`, which frees a DLL that was loaded with `dlopen()`.
- `dlsym()`, which obtains a pointer to an exported function or exported variable.
- `dlerror()`, which returns information about the last DLL failure on this thread that occurred in one of the `dlopen()` family of functions.

While you can use both families of explicit DLL services in a single application, you cannot mix usage across those families. So a handle returned by `dllload()` can be used only with `dllqueryfn()`, `dllqueryvar()`, or `dllfree()`. And a handle returned by `dlopen()` can be used only with `dlsym()` and `dldclose()`.

Since the `dlopen()` family of functions are part of the Single UNIX Specification, Version 3, they should be used in a new application if cross-platform portability is a concern.

For more information about the C runtime services, see *z/OS XL C/C++ Run-Time Library Reference*.

To explicitly call a DLL in your application:

- Determine the names of the exported functions and variables that you want to use. You can get this information from the DLL provider's documentation or by looking at the definition side-deck file that came with the DLL. A definition side-deck is a directive file that contains an `IMPORT` control statement for each function and variable exported by that DLL.

Dynamic Link Libraries (DLLs)

- If you are using the `dllload()` family of functions, include the DLL header file `<dll.h>` in your application. If you are using the `dlopen()` family of functions, include the DLL header file `<dlopen.h>` in your application.
- Compile your source as usual.
- Bind your object with the binder using the same `AMODE` value as the DLL.

Note: You do not need to bind with the definition side-deck if you are calling the DLL explicitly with the runtime services, since there are no references from the source code to function or variable names in the DLL for the binder to resolve. Therefore the DLL will not be loaded until you explicitly load it with the `dllload()` or `dlopen()` runtime service.

Figure 3 and Figure 4 on page 18 are examples of applications that use explicit DLL calls.

Explicit use of a DLL in a C application

The following example shows explicit use of a DLL in a C application.

```
#include <dll.h>
#include <stdio.h>
#include <string.h>

#ifdef __cplusplus
extern "C" {
#endif

    typedef int (DLL_FN)(void);

#ifdef __cplusplus
}
#endif

#define FUNCTION      "FUNCTION"
#define VARIABLE     "VARIABLE"

static void Syntax(const char* progName) {
    fprintf(stderr, "Syntax: %s <DLL-name> <type> <identifier>\n"
        "  where\n"
        "  <DLL-name> is the DLL to load,\n"
        "  <type> can be one of FUNCTION or VARIABLE\n"
        "  and <identifier> is the function or variable\n"
        "  to reference\n", progName);
    return;
}

main(int argc, char* argv[]) {
    int value;
    int* varPtr;
    char* dll;
    char* type;
    char* id;
    dllhandle* dllHandle;

    if (argc != 4) {
        Syntax(argv[0]);
        return(4);
    }
}
```

Figure 3. Explicit use of a DLL in an application using the `dllload()` family of functions (Part 1 of 2)

```

d11 = argv[1];
type = argv[2];
id = argv[3];

d11Handle = d11load(d11);
if (d11Handle == NULL) {
    perror("DLL-Load");
    fprintf(stderr, "Load of DLL %s failed\n", d11);
    return(8);
}

if (strcmp(type, FUNCTION) ) {
    if (strcmp(type, VARIABLE) ) {
        fprintf(stderr,
            "Type specified was not " FUNCTION " or " VARIABLE "\n");
        Syntax(argv[0]);
        return(8);
    }
    /*
     * variable request, so get address of variable
     */
    varPtr = (int*)(d11queryvar(d11Handle, id));
    if (varPtr == NULL) {
        perror("DLL-Query-Var");
        fprintf(stderr, "Variable %s not exported from %s\n", id, d11);
        return(8);
    }
    value = *varPtr;
    printf("Variable %s has a value of %d\n", id, value);
}
else {
    /*
     * function request, so get function descriptor and call it
     */
    DLL_FN* fn = (DLL_FN*) (d11queryfn(d11Handle, id));
    if (fn == NULL) {
        perror("DLL-Query-Fn");
        fprintf(stderr, "Function %s() not exported from %s\n", id, d11);
        return(8);
    }
    value = fn();
    printf("Result of call to %s() is %d\n", id, value);
}
d11free(d11Handle);

return(0);
}

```

Figure 3. Explicit use of a DLL in an application using the `d11load()` family of functions (Part 2 of 2)

Dynamic Link Libraries (DLLs)

```
#define _UNIX03_SOURCE

#include <dlfcn.h>
#include <stdio.h>
#include <string.h>

#ifdef __cplusplus
extern "C" {
#endif

    typedef int (DLL_FN)(void);

#ifdef __cplusplus
}
#endif

#define FUNCTION        "FUNCTION"
#define VARIABLE        "VARIABLE"

static void Syntax(const char* progName) {
    fprintf(stderr, "Syntax: %s <DLL-name> <type> <identifier>\n"
        "   where\n"
        "   <DLL-name> is the DLL to open,\n"
        "   <type> can be one of FUNCTION or VARIABLE,\n"
        "   and <identifier> is the symbol to reference\n"
        "   (either a function or variable, as determined by"
        "   <type>)\n", progName);
    return;
}
```

Figure 4. Explicit Use of a DLL in a application using the dlopen() family of functions (Part 1 of 2)

```

main(int argc, char* argv[]) {
    int value;
    void* symPtr;
    char* dll;
    char* type;
    char* id;
    void* dllHandle;
    if (argc != 4) {
        Syntax(argv[0]);
        return(4);
    }

    dll = argv[1];
    type = argv[2];
    id = argv[3];

    dllHandle = dlopen(dll, 0);
    if (dllHandle == NULL) {
        fprintf(stderr, "dlopen() of DLL %s failed: %s\n", dll, dlerror());
        return(8);
    }

    /*
     * get address of symbol (may be either function or variable)
     */
    symPtr = (int*)(dlsym(dllHandle, id));
    if (symPtr == NULL) {
        fprintf(stderr, "dlsym() error: symbol %s not exported from %s: %s\n",
            id, dll, dlerror());
        return(8);
    }

    if (strcmp(type, FUNCTION)) {
        if (strcmp(type, VARIABLE)) {
            fprintf(stderr,
                "Type specified was not " FUNCTION " or " VARIABLE "\n");
            Syntax(argv[0]);
            return(8);
        }
        /*
         * variable request, so display its value
         */
        value = *(int *)symPtr;
        printf("Variable %s has a value of %d\n", id, value);
    }
    else {
        /*
         * function request, so call it and display its return value
         */
        value = ((DLL_FN *)symPtr)();
        printf("Result of call to %s() is %d\n", id, value);
    }
    dlclose(dllHandle);

    return(0);
}

```

Figure 4. Explicit Use of a DLL in a application using the *dlopen()* family of functions (Part 2 of 2)

For more information on the DLL functions, see *z/OS XL C/C++ Run-Time Library Reference*.

Managing DLLs when running DLL applications

This section describes how Language Environment manages loading, sharing and freeing DLLs when you run a DLL application.

Loading DLLs

When you load an AMODE 64 DLL for the first time, either implicitly or via an explicit `dllload()` or `dlopen()`, writable static area is initialized. If the DLL is written in C++ and contains static objects, then their constructors are run.

You can load DLLs from a z/OS UNIX HFS as well as from conventional data sets. The following list specifies the order of a search for unambiguous and ambiguous file names.

- **Unambiguous file names**

- If the file has an unambiguous HFS name (it starts with a `./` or contains a `/`), the file is searched for only in the HFS.
- If the file has an unambiguous MVS™ name, and starts with two slashes (`//`), the file is only searched for in MVS.

- **Ambiguous file names**

For ambiguous cases, the settings for POSIX are checked.

- When specifying the `POSIX(ON)` runtime option, the runtime library attempts to load the DLL as follows:
 1. An attempt is made to load the DLL from the HFS. This is done using the system service `BPX4LOD`. For more information on this service, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
If the environment variable `LIBPATH` is set, each directory listed will be searched for the DLL. Otherwise the current directory will be searched for the DLL. Note that a search for the DLL in the HFS is case-sensitive.
 2. If the DLL is found and contains an external link name of eight characters or less, the uppercase external link name is used to attempt a `LOAD` from the caller's MVS load library search order. If the DLL is not found or the external link name is more than eight characters, then the load fails.
 3. If the DLL is found and its sticky bit is on, any suffix is stripped off. Next, the name is converted to uppercase, and the base DLL name is used to attempt a `LOAD` from the caller's MVS load library search order. If the DLL is not found or the base DLL name is more than eight characters, the version of the DLL in the HFS is loaded.
 4. If the DLL is found and does not fall into one of the previous two cases, a load from the HFS is attempted.

If the DLL could not be loaded from the HFS because the file was not found or the application doesn't have sufficient authority to search for or read that file (that is, `BPX4LOD` fails with `errno` `ENOENT`, `ENOSYS`, or `EACCESS`), then an attempt is made to load the DLL from the caller's MVS load library search order. For all other failures from `BPX4LOD`, the load of the DLL is terminated.

- For an implicit DLL load, the error is reported with the `errno` and `errnojr` displayed in message `CEE3512S`.
- For an explicit DLL load with `dllload()`, the service returns with the failing `errno` and `errnojr` values set.
- For an explicit DLL load with `dlopen()`, the `dLError()` service will return the failing error.

Correct the indicated error and re-run the application.

If the DLL could not be loaded from the HFS, an attempt is made to load the DLL from the caller's MVS load library search order. This is done by calling the LOAD service with the DLL name, which must be eight characters or less (it will be converted to uppercase). LOAD searches for it in the following sequence:

1. Runtime library services (if active)
2. Job pack area (JPA)
3. TASKLIB
4. STEPLIB or JOBLIB. If both are allocated, the system searches STEPLIB and ignores JOBLIB.
5. LPA
6. Libraries in the linklist

For more information, see *z/OS MVS Initialization and Tuning Guide*

- When POSIX(OFF) is specified the sequence is reversed.
 - An attempt to load the DLL is made from the caller's MVS load library search order.
 - If the DLL could not be loaded from the caller's MVS load library then an attempt is made to load the DLL from the HFS.

Recommendation: All DLLs used by an application should be referred to by unique names, whether ambiguous or not. Using multiple names for the same DLL (for example, aliases or symlinks) may result in a decrease in DLL load performance. The use of HFS symbolic links by themselves will not degrade performance, as long as the application refers to the DLL solely through the symbolic link name. To help ensure this, when building an application with implicit DLL references always use the same side deck for each DLL. Also, make sure that explicit DLL references with `dllload()` specify the same DLL name (case matters for HFS loads).

Changing the search order for DLLs while the application is running (for example, changing LIBPATH) may result in errors if ambiguous file names are used.

Sharing DLLs

DLLs are shared at the enclave level (as defined by Language Environment). A referenced DLL is loaded only once per enclave and only one copy of the writable static is created or maintained per DLL per enclave. Thus, one copy of a DLL serves all modules in an enclave regardless of whether the DLL is loaded implicitly or explicitly. A copy is implicit through a reference to a function or variable. A copy is explicit through a DLL load. You can access the same DLL within an enclave both implicitly and by explicit runtime services.

All accesses to a variable in a DLL in an enclave refer to the single copy of that variable. All accesses to a function in a DLL in an enclave refer to the single copy of that function.

Although only one copy of a DLL is maintained per enclave, multiple logical loads are counted and used to determine when the DLL can be deleted. For a given DLL in a given enclave, there is one logical load for each explicit `dllload()` or `dlopen()` request. DLLs that are referenced implicitly may be logically loaded at application initialization time if the application references any data exported by the DLL, or the logical load may occur during the first implicit call to a function exported by the DLL.

Freeing DLLs

You can free explicitly loaded DLLs with a `dllfree()` or `dllclose()` request. This request is optional because the DLLs are automatically deleted by the runtime library when the enclave is terminated.

Implicitly loaded DLLs cannot be deleted from the DLL application code. They are deleted by the runtime library at enclave termination. Therefore, if a DLL has been both explicitly and implicitly loaded, the DLL can only be deleted by the runtime when the enclave is terminated.

Creating a DLL or a DLL application

Building an AMODE 64 DLL or a DLL application is similar to creating a C or C++ application. It involves the following steps:

1. Writing your source code
2. Compiling your source code
3. Binding your object modules

See *z/OS XL C/C++ Programming Guide* for additional language-specific details.

Building a DLL

This section shows how to build a DLL. See “Building a DLL application” on page 27 for information about building a DLL application.

Writing DLL code

Writing your C DLL code

To build a C DLL, write code using the `#pragma export` directive to export specific external functions and variables as shown in Figure 5.

```
#pragma export(bopen)
#pragma export(bclose)
#pragma export(bread)
#pragma export(bwrite)
int bopen(const char* file, const char* mode) {
    ...
}
int bclose(int) {
    ...
}
int bread(int bytes) {
    ...
}
int bwrite(int bytes) {
    ...
}
#pragma export(berror)
int berror;
char buffer[1024];
...
```

Figure 5. Using `#pragma export` to Create a DLL Executable Module Named BASICIO

For the previous example, the functions `bopen()`, `bclose()`, `bread()`, and `bwrite()` are exported; the variable `berror` is exported; and the variable `buffer` is not exported.

Note: To export **all** defined functions and variables with external linkage in the compilation unit to the users of the DLL, compile with the `EXPORTALL` compile option. All defined functions and variables with external linkage will be accessible from this DLL and by all users of this DLL. However, exporting all functions and variables has a performance penalty, especially when compiling with the C/C++ IPA option. When you use `EXPORTALL` you do not need to include `#pragma export` in your code.

Writing your C++ DLL code

To create a C++ DLL:

- Ensure that classes and class members are exported correctly, especially if they use templates.
- Use `_Export` or the `#pragma export` directive to export specific functions and variables.

For example, to create a DLL executable module `TRIANGLE`, export the `getarea()` function, the `getperim()` function, the static member `objectCount` and the static constructor for class `triangle` using `#pragma export`:

```
class triangle : public area
{
    public:
        static int objectCount;
        getarea();
        getperim();
        triangle::triangle(void);
};
#pragma export(triangle::objectCount)
#pragma export(triangle::getarea())
#pragma export(triangle::getperim())
#pragma export(triangle::triangle(void))
```

Figure 6. Using `#pragma Export` to Create a DLL Executable Module `TRIANGLE`

- Do not inline the function if you apply the `_Export` keyword to the function declaration.

```
class triangle : public area
{
    public:
        static int _Export objectCount;
        double _Export getarea();
        double _Export getperim();
        _Export triangle::triangle(void);
};
```

Figure 7. Using `_export` to Create DLL Executable Module `TRIANGLE`

- Always export static constructors and destructors when using the `_Export` keyword.
- Apply the `_Export` keyword to a class. This keyword automatically exports static members and defined functions of that class, constructors, and destructors.

Dynamic Link Libraries (DLLs)

```
class Export triangle
{
public:
    static int objectCount;
    double getarea();
    double getperim();
    triangle::triangle(void);
};
```

- To export all external functions and variables in the compilation unit to the users of this DLL, you can also use the compiler option EXPORTALL. This compiler option is described in *z/OS XL C/C++ User's Guide*, and #pragma export directives are described in detail in *z/OS XL C/C++ Language Reference*. If you use the EXPORTALL option, you do not need to include #pragma export or _Export in your code.

Writing your Language Environment-conforming AMODE 64 assembler DLL code

To build an assembler DLL, your assembler routine must conform to Language Environment conventions for AMODE 64 applications. To do this, begin by using the Language Environment macros CELQPRLG and CELQEPLG. The EXPORT= keyword parameter on the CELQPRLG macro allows you to identify specific assembler entry points for export. The CEEPDDA macro allows you to define data in your assembler routine that can be exported. Details on all Language Environment assembler macros are in Chapter 21, "Assembler Considerations," on page 149.

Figure 8 on page 25 shows how to use Language Environment macros to create an AMODE 64 assembler DLL. The CELQPRLG prolog macro has EXPORT=YES specified to mark this entry point exported. In this particular case we want the exported function known externally in lower case, so the CELQPRLG macro is followed by an assembler ALIAS statement. The ALIAS can be used to "name" the exported function with a mixed-case name up to 256 characters long. This assembler DLL also has two exported variables, "DllVar64" (initial value = 123) and "DllStr64" (initial value is the C string "Hello World"). When the exported function "dllfunc64" is called, it sets "DllVar64" to 456 and truncates the "DllStr64" C string to "Hello".

For more information on macros, see Chapter 21, "Assembler Considerations," on page 149.

```

DLLFUNC CELQPRG EXPORT=YES,PSECT=ADLA6EVP
DLLFUNC ALIAS C'dllfunc64'
*      Symbolic Register Definitions and Usage
R3     EQU    3          RETURN VALUE
R5     EQU    5          ENVIRONMENT
R6     EQU    6          ENTRY POINT ADDRESS
R7     EQU    7          RETURN POINT ADDRESS
R8     EQU    8          Work register
R9     EQU    9          Work register
R15    EQU    15         Entry point address
*
      WTO    'ADLL6EV2: Exported function dllfunc64 entered',ROUTCDE=11
*
      WTO    'ADLL6EV2: Setting D11Var64 to 456',ROUTCDE=11

```

Figure 8. Using Language Environment macros to create an AMODE 64 assembler DLL executable named ADLL6EV2 (Part 1 of 2)

```

*
      CEEPLDA D11Var64,REG=9
      LA     R8,456
      ST     R8,0(R9)
*
      WTO    'ADLL6EV2: Truncating exported string to "Hello"',      X
      ROUTCDE=11
*
      CEEPLDA D11Str64,REG=9
      LA     R8,0
      STC    R8,5(R9)
*
      WTO    'ADLL6EV2: Done.',ROUTCDE=11
*
      SR     R3,R3
RETURN  DS    0H
      CELQEPLG
*
      CEEPDDA D11Var64,SCOPE=EXPORT
      DC     A(123)
      CEEPDDA END
      CEEPDDA D11Str64,SCOPE=EXPORT
      DC     C'Hello World'
      DC     X'00'
      CEEPDDA END
*
      LTRG
CEEDSAHP CEEDSA SECTYPE=XPLINK
      CECAA
*
      END     DLLFUNC

```

Figure 8. Using Language Environment macros to create an AMODE 64 assembler DLL executable named ADLL6EV2 (Part 2 of 2)

Compiling your DLL code

For C or C++ source, compile with the LP64 compiler option. There is no additional option when compiling with LP64 to generate “DLL-enabled code.” All AMODE 64 C and C++ code is automatically enabled for DLLs.

Dynamic Link Libraries (DLLs)

Note: DLLs must be reentrant; you must use the RENT C compiler option (C++ is always reentrant).

For assembler source, you must use the G0FF option.

Binding your DLL code

Use the DLL support in the DFSMS binder, for linking AMODE 64 DLL applications. Note that binder-based AMODE 64 DLLs must reside in PDSEs, rather than PDS data sets. When binding a DLL application using the DFSMS binder, the following binder externals are used:

- The binder option CASE(MIXED) is required when binding DLLs that use mixed-case exported names.
- The binder options RENT, DYNAM(DLL), and COMPAT(PM4) or COMPAT(CURRENT) are required.
- When binding a DLL, a SYSDEFSD DD statement must be specified, indicating the data set where the binder should create a DLL definition side-deck. The DLL definition side-deck contains IMPORT control statements for each of the symbols exported by a DLL. If you are using z/OS UNIX, specify the following option for the bind step for c89 or the c++ command:

```
-W 1,DLL
```

- The binder SYSLIN input, the binding code that references DLL code, must include the DLL definition side-decks for the DLLs that are to be dynamically referenced from the module being bound. See *z/OS MVS Program Management: User's Guide and Reference* for further details.

Binding C

When binding the C object module as shown in Figure 5 on page 22, the binder generates the following definition side-deck:

```
IMPORT CODE64 'BASICIO'   bopen
IMPORT CODE64 ,BASICIO,  bclose
IMPORT CODE64 ,BASICIO,  bread
IMPORT CODE64 ,BASICIO,  bwrite
IMPORT DATA64 ,BASICIO, berror
```

You can edit the definition side-deck to remove any functions or variables that you do not want to export. For instance, in the above example, if you do not want to expose berror, remove the control statement IMPORT DATA64 ,BASICIO, berror from the definition side-deck.

Notes:

1. You should also provide a header file containing the prototypes for exported functions and external variable declarations for exported variables.
2. Side-decks are created without newline characters, therefore you cannot edit them with an editor that expects newline characters, such as vi in z/OS UNIX.

For more information on binding C, see *z/OS XL C/C++ User's Guide*.

Binding C++

When binding the C++ object modules shown in Figure 6 on page 23, the binder generates the following definition side-deck.

```
IMPORT CODE64 ,TRIANGLE, getarea__8triangleFv
IMPORT CODE64 ,TRIANGLE, getperim__8triangleFv
IMPORT CODE64 ,TRIANGLE, __ct__8triangleFv
```

You can edit the definition side-deck to remove any functions and variables that you do not want to export. In the above example, if you do not want to expose `getperim()`, remove the control statement `IMPORT CODE64 ,TRIANGLE, getperim__8triangleFv` from the definition side-deck.

Notes:

1. Removing functions and variables from the definition side-deck does not minimize the performance impact caused by specifying the `EXPORTALL` compiler option.
2. Side-decks are created without newline characters, therefore you cannot edit them with an editor that expects newline characters, such as `vi` in `z/OS UNIX`.

The definition side-deck contains mangled names, such as `getarea__8triangleFv`. To find the original function or variable name in your source module, review the compiler listing created or use the `CXXFILT` utility. This will permit you to see both the mangled and demangled names. For more information on the `CXXFILT` utility, and on binding `C++`, see *z/OS XL C/C++ User's Guide*.

Binding assembler

When binding the assembler object module as shown in Figure 8 on page 25, the binder generates the following definition side-deck:

```
IMPORT CODE64, 'ADLL6EV2', 'd11func64'
IMPORT DATA64, 'ADLL6EV2', 'D11Str64'
IMPORT DATA64, 'ADLL6EV2', 'D11Var64'
```

Building a DLL application

The DLL application may consist of multiple source modules. Some of the source modules may contain references to imported functions, imported variables, or both.

To use a load-on-call DLL in your DLL application, perform the following steps:

Writing your DLL application code

Write your code as you would if the functions were statically bound. Assembler code that will access imported functions and/or imported variables must use the Language Environment macros.

Compiling your DLL application code

- Compile your C source files with the following compiler options:
 - LP64
 - RENT
 - LONGNAME

These options instruct the compiler to generate special code when calling functions and referencing external variables.

- Compile your C++ source files with the LP64 compiler option.
- Assembler DLL application source files must be assembled with the `G0FF` option.

Binding your DLL application code

- The binder option `CASE(MIXED)` is required when binding DLL applications that use mixed-case exported names.
- The binder options `RENT`, `DYNAM(DLL)`, and `COMPAT(PM4)` or `COMPAT(CURRENT)` are required.

Include the definition side-deck from the DLL provider in the set of object modules to bind. The binder uses the definition side-deck to resolve references

Dynamic Link Libraries (DLLs)

to functions and variables defined in the DLL. If you are referencing multiple DLLs, you must include multiple definition side-decks.

Note: Because definition side-decks in automatic library call (autocall) processing will not be resolved, you must use the INCLUDE statement.

After final autocall processing of DD SYSLIB is complete, all DLL-type references that are not statically resolved are compared to IMPORT control statements. Symbols on IMPORT control statements are treated as definitions, and cause a matching unresolved symbol to be considered dynamically rather than statically resolved. A dynamically resolved symbol causes an entry in the binder B_IMPEXP to be created. If the symbol is unresolved at the end of DLL processing, it is not accessible at run time.

Addresses of statically bound symbols are known at application load time, but addresses of dynamically bound symbols are not. Instead, the runtime library that loads the DLL that exports those symbols finds their addresses at application run time. The runtime library also fixes up the importer's linkage blocks (descriptors) in C_WSA64 during program execution.

The following code fragment illustrates how a C++ application can use the TRIANGLE DLL described previously (see "Writing your C++ DLL code" on page 23). Compile normally and bind with the definition side-deck provided with the TRIANGLE DLL.

```
extern int getarea(); /* function prototype */
main () {
    ...
    getarea();      /* imported function reference */
    ...
}
```

The following code fragment illustrates how an assembler routine can use the ADLL6EV2 DLL described previously (see "Writing your Language Environment-conforming AMODE 64 assembler DLL code" on page 24). Assemble and bind with the definition side-deck provided with the ADLL6EV2 DLL.

```

DLLAPPL  CELQPRLG PSECT=ADLA6IFP
*
R3      EQU  3          RETURN VALUE
R5      EQU  5          ENVIRONMENT
R6      EQU  6          ENTRY POINT ADDRESS
R7      EQU  7          RETURN POINT ADDRESS
R8      EQU  8          WORK REGISTER
R9      EQU  9          WORK REGISTER
*
      WTO  'ADLA6IV4: Calling imported function dllfunc64',ROUTCDE=11
*
      CELQCALL  dllfunc64,WORKREG=10
*
      WTO  'ADLA6IV4: Getting address of imported var D11Var64',  X
      ROUTCDE=11
*
      CEEPLDA  D11Var64,REG=9
*
* Set value of imported variable to 789
*
      LA  R8,789
      ST  R8,0(,R9)
*
      WTO  'ADLA6IV4: Done.',ROUTCDE=11
*
      SR  R3,R3
RETURN  DS  0H
      CELQEPLG
*
      CEEPDDA  D11Var64,SCOPE=IMPORT
      LTORG
CEEDSAHP CEEDSA SECTYPE=XPLINK
      CEECAA
*
      END      DLLAPPL

```

Figure 9. Assembler DLL Application calling an assembler DLL

See Figure 10 on page 30 for a summary of the processing steps required for the application (and related DLLs).

Creating and using DLLs

Figure 10 on page 30 summarizes the use of DLLs for both the DLL provider and for the writer of applications that use them. In this example, application ABC is referencing functions and variables from two DLLs, XYZ and PQR. The connection between DLL preparation and application preparation is shown. Each DLL shown contains a single compilation unit. The same general scheme applies for DLLs composed of multiple compilation units, except that they have multiple compiles and a single bind for each DLL. For simplicity, this example assumes that ABC does not export variables or functions and that XYZ and PQR do not use other DLLs.

Dynamic Link Libraries (DLLs)

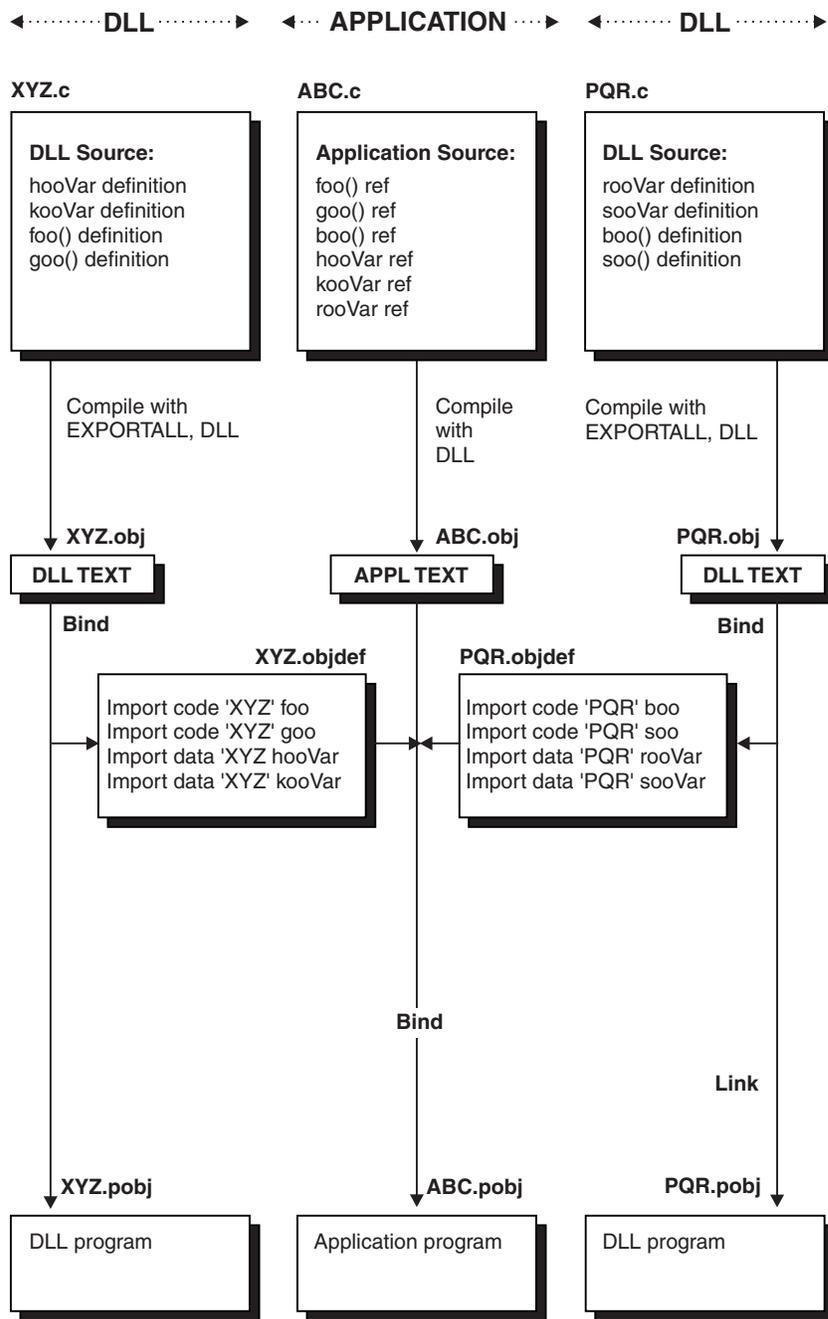


Figure 10. Summary of DLL and DLL Application Preparation and Usage

DLL restrictions

Consider the following restrictions when creating DLLs and DLL applications:

- The AMODE of a DLL application must be the same as the AMODE of the DLL that it calls. For the case described here, that means both the DLL and DLL application must be AMODE 64. There is no support for mixing AMODE 64 with either AMODE 31/AMODE 24 applications.

- DLLs must be REENTRANT. Be sure to specify the RENT option when you bind your code. Unpredictable results will occur if you bind a DLL as NORENT. One possible symptom you may see that indicates the DLL was bound as NORENT is more than one writable static area for the same DLL.
- In a C/C++ DLL application that contains `main()`, `main()` cannot be exported.
- In C++ applications, you cannot implicitly or explicitly perform a physical load of a DLL while running static destructors. However, a logical load of a DLL (meaning that the DLL has previously been loaded into the enclave) is allowed from a static destructor. In this case, references from the load module containing the static destructor to the previously-loaded DLL are resolved.
- You cannot use the C functions `set_new_handler()` or `set_unexpected()` in a DLL if the DLL application is expected to invoke the new handler or unexpected function routines.
- When using the explicit C DLL functions in a multithreaded environment, avoid any situation where one thread frees a DLL while another thread calls any of the DLL functions. For example, this situation occurs when a `main()` function uses `dllload()` or `dlopen()` to load a DLL, and then creates a thread that uses the `ftw()` function. The `ftw()` target function routine is in the DLL. If the `main()` function uses `dllfree()` or `dllclose()` to free the DLL, but the created thread uses `ftw()` at any point, you will get an abend.
To avoid a situation where one thread frees a DLL while another thread calls a DLL function, do either of the following:
 - Do not free any DLLs by using `dllfree()` or `dllclose()` (Language Environment will free them when the enclave is terminated).
 - Have the `main()` function call `dllfree()` or `dllclose()` only after all threads have been terminated.
- For C/C++ DLLs to be processed by IPA, they must contain at least one function or method. Data-only DLLs will result in a compilation error.
- The use of circular C++ DLLs may result in unpredictable behavior related to the initialization of non-local static objects. For example, if a static constructor (being run as part of loading DLL "A") causes another DLL "B" to be loaded, then DLL "B" (or any other DLLs that "B" causes to be loaded before static constructors for DLL "A" have completed) cannot expect non-local static objects in "A" to be initialized (that is what static constructors do). You should ensure that non-local static objects are initialized before they are used, by coding techniques such as counters or by placing the static objects inside functions.

Improving performance

This section contains some hints on using DLLs efficiently. Effective use of DLLs may improve the performance of your application.

- If you are using a particular DLL frequently across multiple address spaces, the DLL can be installed in dynamic LPA. Installing in dynamic LPA may give you the performance benefits of a single rather than multiple load of the DLL.
- Group external variables into one external structure.
- When using z/OS UNIX avoid unnecessary load attempts.

Language Environment supports loading a DLL residing in the HFS or a data set. However, the location from which it tries to load the DLL first varies depending whether your application runs with the runtime option `POSIX(ON)` or `POSIX(OFF)`.

If your application runs with `POSIX(ON)`, Language Environment tries to load the DLL from the HFS first. If your DLL is a data set member, you can avoid

Dynamic Link Libraries (DLLs)

searching the HFS directories. To direct a DLL search to a data set, prefix the DLL name with two slashes (//) as is in the following example:

```
//MYDLL
```

If your application runs with `POSI(0FF)`, Language Environment tries to load your DLL from a data set. If your DLL is an HFS file, you can avoid searching a data set. To direct a DLL search to the HFS, prefix the DLL name with a period and slash (./) as is done in the following example.

```
./mydll
```

Note: DLL names are case sensitive in the HFS. If you specify the wrong case for your DLL that resides in the HFS, it will not be found.

- For C/C++ IPA, you should only export subprograms (functions and C++ methods) or variables that you need for the interface to the final DLL. If you export subprograms or variables unnecessarily (for example, by using the `EXPORTALL` option), you severely limit IPA optimization. In this case, global variable coalescing and pruning of unreachable or 100% inlined code does not occur. To be processed by IPA, DLLs must contain at least one subprogram. Attempts to process a data-only DLL will result in a compilation error.

Chapter 4. Binding, loading, and running under batch

You process an AMODE 64 application under batch by submitting batch jobs to the operating system. A job might consist of one or more of the following job steps:

- Compiling a program
- Binding an application
- Running an application

IBM-supplied cataloged procedures allow you to compile, bind or load, and run an application without supplying all the job control language (JCL) required for a job step. For information about cataloged procedures for AMODE 64 applications, see *z/OS XL C/C++ User's Guide*. If the statements in the cataloged procedures do not match your requirements exactly, you can modify them or add new statements for the duration of a job.

The following section provides an overview of binding, loading, and running Language Environment-conforming applications under batch. For detailed information about binding, see *z/OS MVS Program Management: User's Guide and Reference*.

z/OS UNIX has its own section on binding, loading, and running C applications (see Chapter 5, "Binding and executing AMODE 64 programs using z/OS UNIX System Services," on page 41).

Basic binding and running under batch

This section describes how to accept and to override the default Language Environment run-time options under MVS.

Specifying runtime options in the EXEC statement

You can pass runtime options by using the PARM= parameter in your JCL. The general form for specifying runtime options in the PARM parameter of the EXEC statement is:

```
//[stepname] EXEC PGM=program_name,  
//          PARM='[runtime options/][program parameters]'
```

For example, if you want to generate a storage report and runtime options report for an AMODE 64 program named PROGRAM1, specify the following:

```
//G01 EXEC PGM=PROGRAM1,PARM='RPTSTG(ON),RPTOPTS(ON)'
```

The runtime options that are passed to the main routine must be followed by a slash (/) to separate them from program parameters. For HLL considerations to keep in mind when specifying runtime options, see "Specifying runtime options and program arguments" on page 50. The EXECOPS option for C and C++ is used to specify that runtime options passed as parameters at execution time are to be processed by Language Environment. The option NOEXECOPS specifies that runtime options are not to be processed from execution parameters and are to be treated as program parameters.

Running under Batch

For z/OS XL C/C++, a user can specify either EXECOPS or NOEXECOPS in a #pragma runopts directive or as a compiler option. EXECOPS is the default for z/OS XL C/C++. When EXECOPS is in effect, you can pass runtime options in the EXEC statement in your JCL.

Specifying Run-Time Options with the CEEOPTS DD card

Language Environment supports the ability to provide additional run-time options through a DD card. The name of the DD must be CEEOPTS. The DD must be available during initialization of the "enclave" so that the options can be merged.

The "Last Where Set" column of the Language Environment Run-Time Options report uses DD:CEEOPTS to indicate that CEEOPTS was the last time this option was set.

The CEEOPTS DD is ignored for programs invoked using one of the exec() family of functions.

Providing bind input

Input to the bind process can be:

- One or more object modules
- Control statements for the bind process
- Previously bound AMODE 64 executable programs you want to combine into a single AMODE 64 executable module
- A DLL side-deck if your application implicitly references DLL functions or data

Figure 11 shows the basic batch bind process for your application.

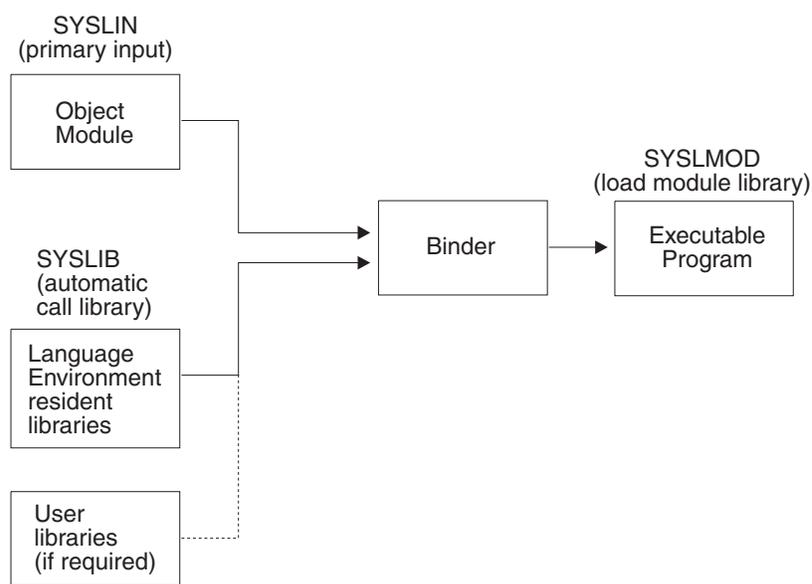


Figure 11. Basic Batch bind Processing

Writing JCL for the bind process

You can use cataloged procedures rather than supply all the JCL required for a job step. You can use JCL statements to override the statements of the cataloged procedure to tailor the information provided by the bind process.

For a description of the IBM-supplied cataloged procedures that include a bind step, see *z/OS XL C/C++ User's Guide*.

- Invoking with the EXEC Statement

Use the EXEC job control statement in your JCL to invoke the binder. The EXEC statement is:

```
//LKED EXEC PGM=IEWL
```

- Using the PARM Parameter

Use the PARM parameter of the EXEC job control statement to select one or more of the optional facilities provided by the binder. For example, if you want a mapping of the AMODE 64 executable program produced by the bind process, specify:

```
//LKED EXEC PGM=IEWL,PARM='MAP'
```

For a description of bind options, see “Bind options” on page 38.

- Required DD Statements

The bind process requires three standard data sets. You must define these data sets in DD statements with the ddnames SYSLIN, SYSLMOD, and SYSPRINT. The required data sets and their characteristics are shown in Table 4.

Table 4. Required Data Sets Used for binding

ddname	Type	Function
SYSLIN	Input	Primary input to the bind process consists of a sequential data set, members from a PDS or PDSE, or an in-stream data set. The primary input must be composed of one or more separately compiled object modules or bind control statements. An executable program cannot be part of the primary input, although it can be introduced by the INCLUDE control statement (see “Using the INCLUDE statement” on page 37).
SYSLMOD	Output	The data set where output (executable program) from the bind process is stored. It must be a PDSE or a file in the HFS (using PATH=).
SYSPRINT	Output	SYSPRINT defines the location for the listing that includes reference tables for the executable program. Output from the bind process: <ul style="list-style-type: none"> • Diagnostic messages • Informational messages • Module map • Cross-reference list

- Optional DD Statements

If you want to use the automatic call library, you must define a data set using a DD statement with the name SYSLIB. You can also specify additional data sets containing object modules and AMODE 64 executable programs as additional input to the bind process. These data set names and their characteristics are shown in Table 5 on page 36.

Table 5. Optional Data Sets Used for binding

ddname	Type	Function
SYSLIB ¹	Library	<p>Secondary input to the binder consists of object modules or load modules that are included in the executable program from the automatic call library. The automatic call library contains load modules or object modules that are used as secondary input to the binder to resolve external symbols left undefined after all the primary input has been processed. The automatic call library can include:</p> <ul style="list-style-type: none"> • Libraries that contain object modules, with or without binder control statements • Libraries that contain executable programs • The libraries that contain the Language Environment resident routines. For a description of this data set see “Planning to bind and run” on page 7). <p>SYSLIB is input to the binder only if the CALL=NO bind option is not in effect (see Table 6 on page 38, in <i>z/OS MVS Program Management: User’s Guide and Reference</i>, or <i>z/OS TSO/E Command Reference</i> for more information). You can also identify secondary input to the binder with the INCLUDE statement.</p> <p>A routine compiled with a Language Environment-conforming compiler cannot be executed until the appropriate Language Environment resident routines have been linked into the executable program. The Language Environment resident routines are contained in the SCEEBND2 library; the data set name could be CEE.SCEEBND2. If you are unsure where SCEEBND2 has been installed at your location, contact your system administrator. This data set must be specified in the SYSLIB statement in your JCL.</p> <p>In the following example, the SYSLIB DD statement is written so that Language Environment resident library routines are included as secondary input into your executable program:</p> <pre>//SYSLIB DD DSN=CEE.SCEEBND2,DISP=SHR</pre>
User-specified ²	Input	You can use ddnames to get additional executable programs and object modules.

Notes:

- ¹ Required for library run-time routines
² Optional data set

• Examples of bind JCL

A typical sequence of job control statements for binding an object module (compiled with LP64) into an AMODE 64 executable program is shown in Figure 12 on page 37. The NAME binder control statement in the figure puts PROGRAM1 in USER.LOADLIB with the member name PROGRAM1.

```

//LKED64X EXEC PGM=IEWL,REGION=20M,
//          PARM='AMODE=64,RENT,DYNAM=DLL,CASE=MIXED,MAP,LIST=NOIMP'
//SYSPRINT DD  SYSOUT=*
//SYSLMOD DD  DSNAME=USER.PDSELIB,UNIT=SYSALLDA,
//          DISP=(NEW,KEEP),SPACE=(TRK,(7,7,1)),DSNTYPE=LIBRARY
//SYSLIB DD  DSNAME=CEE.SCEEBND2,DISP=SHR
//SYSLIN DD  DSNAME=USER.OBJLIB(PROGRAM1),DISP=SHR
//          DD  DSNAME=CEE.SCEELIB(CELQS003),DISP=SHR
//SYSDEFSD DD  DUMMY
//SYSIN DD  *
           ENTRY CELQSTRT
           NAME PROGRAM1(R)
/*

```

Figure 12. Creating an AMODE 64 Executable Program under Batch

- Adding Members to a Library

The output from the binder is usually placed in a private program library.

The automatic call library that is used as input to the binder can be a Language Environment library (SCEEBND2 for AMODE 64 applications), a compiler library, a private program library, or a subroutine library.

When you are adding a member to a library, you must specify the member name as follows:

- When a single module is produced as output from the binder, the member name can be specified as part of the data set name in the SYSLMOD.
- When more than one module is produced as output from the binder, the member name for each module must be specified in the NAME option or the NAME control statement. The member name cannot be specified as part of the data set name.

Binder control statements

The following sections describe when and how to use the INCLUDE and LIBRARY control statements with the binder.

Using the INCLUDE statement

Use the INCLUDE control statement to specify additional object modules or AMODE 64 executable programs that you want included in the output executable program. Figure 13 contains an example of how to bind the CELQUOPT CSECT with your application. In the example, CELQUOPT is used to establish application runtime option defaults; see Chapter 6, “Using runtime options,” on page 47 for more information.

```

//SYSLIB DD DSNAME=CEE.SCEEBND2,DISP=SHR
//SYSLIN DD DSNAME=USER.OBJLIB(PROGRAM1),DISP=SHR
//          DD DDNAME=SYSIN
//SYSIN DD *
           INCLUDE SYSLIB(CELQUOPT)
           :
           :
/*

```

Figure 13. Using the INCLUDE binder Control Statement

Using the LIBRARY statement

Use the LIBRARY statement to direct the binder to search a library other than that specified in the SYSLIB DD statement. This method resolves only external references listed on the LIBRARY statement. All other unresolved external references are resolved from the library in the SYSLIB DD statement.

In Figure 14 the LIBRARY statement is used to resolve the external reference PROGRAM2 from the library described in the TESTLIB DD statement.

```
//SYSLIN DD DSN=USER.OBJLIB(PROGRAM1),DISP=SHR
//      DD DDNAME=SYSIN
//TESTLIB DD DSN=USER.TESTLIB,DISP=SHR
//SYSIN  DD *
        LIBRARY TESTLIB(PROGRAM2)
:
/*
```

Figure 14. Using the LIBRARY binder Control Statement

Data sets specified by the INCLUDE statement are incorporated as the binder encounters the statement. In contrast, data sets specified by the LIBRARY statement are used only when there are unresolved references after all the other input is processed.

Bind options

SYSLMOD and SYSPRINT are the data sets used for output. The output varies, depending on the options you select, as shown in Table 6. The underlined options are the defaults.

Table 6. Selected bind options

Option	Function
XREF <u>NOXREF</u>	Specifies if a cross-reference list of data variables is generated.
LIST <u>NOLIST</u>	Specifies if a listing of the bind control statements is generated
NCAL <u>CALL</u>	Specifies if the automatic library call mechanism should be used to locate the modules referred to by the executable program being processed. Use the NCAL command to suppress resolution of external differences. If you do not specify NCAL, the automatic call library mechanism is used to locate the modules referred to by the executable program being processed. Do not use NCAL if your application calls external routines that need to be resolved by an automatic library call.
<u>PRINT</u> NOPRINT	Specifies if bind messages are written on the data set defined by the SYSLOUT DD statement.
MAP <u>NOMAP</u>	Specifies if a map of the load modules is generated and placed in the PRINT data set.
<u>RENT</u> NORENT	Specifies if a module is reentrantable, that is it can be executed by more than one task at a time. A task may begin executing the module before a previous task has completed execution. See Chapter 8, "Making Your Application Reentrant," on page 63 for additional information.

You always receive diagnostic and informational messages as the result of binding, even if you do not specify any options. You can get the other output items by specifying options in the PARM parameter of the EXEC statement in your JCL for binding. See “Writing JCL for the bind process” on page 34 for more information.

For more information about bind options, see *z/OS MVS Program Management: User's Guide and Reference*.

Running an AMODE 64 application under batch

Under batch, you can request the execution of an AMODE 64 executable program in an EXEC statement in your JCL. The EXEC statement marks the beginning of each step in a job or procedure, and identifies the executable program or cataloged procedure that executes.

The general form of the EXEC statement is:

```
//[stepname] EXEC PGM=program_name
```

The *program_name* is the name of the member or alias of the program to be executed. The specified program must be one of the following:

- An executable program that is a member of a private library specified in a STEPLIB DD statement in your JCL.
- An executable program that is a member of a private library specified in a JOBLIB DD statement in your JCL.
- An executable program that has been loaded into shared system storage using dynamic LPA.
- An executable program that is a member of a system library. Examples of system libraries are SYS1.LINKLIB and libraries specified in the LNKLST.

Unless you have indicated that the executable program is in a private library, it is assumed that the executable program is in a system library and the system libraries are searched for the name you specify.

Program library definition and search order

You can define the library in a DD statement in the following ways:

- With the ddname STEPLIB at any point in the job step. The STEPLIB is searched before any system library or JOBLIB specified in a JOBLIB DD statement for the job step in which it appears (although an executable program can also be passed to subsequent job steps in the usual way). When a STEPLIB and JOBLIB are both present, the STEPLIB is searched for the step in which it appears and, for that step, the JOBLIB is ignored.

The system searches for executable programs in the following order of precedence:

1. Library specified in STEPLIB statement
2. Library specified in JOBLIB statement
3. LPA or ELPA
4. The system library SYS1.LINKLIB and libraries concatenated to it through the active LNKLSTxx member of SYS1.PARMLIB

In the following example, the system searches USER.PDSELIB for the routine PROGRAM1 and USER.PDSELIB2 for the routine PROGRAMA:

```
//JOB8 JOB DAVE,MSGLEVEL=(2,0)
//STEP1 EXEC PGM=PROGRAM1
//STEPLIB DD DSNAME=USER.PDSELIB,DISP=SHR
```

Running under Batch

```
//*  
//STEP2 EXEC PGM=PROGRAMA  
//STEPLIB DD DSN=USER.PDSELIB2,DISP=SHR
```

- With the ddname JOBLIB immediately after the JOB statement in your JCL. This library is searched before the system libraries. If any AMODE 64 executable program is not found in the JOBLIB, the system looks for it in the system libraries.

In the following example, the system searches the private library USER.PDSELIB for the member PROGRAM1, reads the member into storage, and executes it.

```
//JOB8 JOB DAVE,MSGLEVEL=(2,0)  
//JOBLIB DD DSN=USER.PDSELIB,DISP=SHR  
//STEP1 EXEC PGM=PROGRAM1
```

Specifying runtime options

Each time your application runs, a set of runtime options must be established. These options determine many of the properties of how the application runs, including its performance, error handling characteristics, storage management, and production of debugging information. Under batch, you can specify run-time options in any of the following places (for additional information about the ways to specify runtime options, see “Specifying runtime options” on page 47):

- In the CELQDOPT CSECT, where the installation default options are located (for more information, see “CEEXOPT invocation syntax” on page 51).
- In the CELQROPT CSECT, where region-wide default options are located (for more information, see “CEEXOPT invocation syntax” on page 51).
- In the CELQUOPT CSECT where user-supplied default options are located (for more information, see “CEEXOPT invocation syntax” on page 51).
- In the CEEPRMxx parmlib member (for more information, see “Parmlib Syntax” on page 54).
- #pragma runopts in C/C++ source code (for more information, see “Specifying runtime options” on page 47).
- In the PARM parameter of the EXEC statement in your JCL.
- In z/OS on the GPARM parameter of the IBM-supplied cataloged procedure (for more information, see *z/OS XL C/C++ User's Guide*).
- In the _CEE_RUNOPTS environment variable, when your application is running under z/OS UNIX and is invoked by one of the exec or spawn family of functions.

Chapter 5. Binding and executing AMODE 64 programs using z/OS UNIX System Services

The interface to the binder for z/OS UNIX System Services (z/OS UNIX) C applications is the z/OS UNIX `c89` utility or the `cc` command, and for C++ applications it is the `c++` command. You can use them to compile and bind a z/OS UNIX C/C++ program in one step, or bind application object modules after the compilation. You must, however, invoke one of the z/OS UNIX shells before you can run the `c89` utility. For more information on using the utility and these commands, see *z/OS UNIX System Services Command Reference*.

For more information on compiling your XL C/C++ applications, see *z/OS XL C/C++ User's Guide*.

Basic binding and running C/C++ applications under z/OS UNIX

z/OS UNIX supports the following environments for running your z/OS UNIX C/C++ AMODE 64 applications:

- z/OS UNIX shells
- TSO/E
- Batch
- z/OS UNIX shells through MVS batch

Using the z/OS UNIX-supplied `c89` utility or `cc` command or `c++` command, you can compile and bind a z/OS UNIX C/C++ application in one step, or bind application object modules separately. To produce an executable file, invoke `c89` and pass it object modules (*file.o* HFS files or *//file.OBJ* MVS data sets) without using the `-c` option.

See *z/OS UNIX System Services Command Reference* for information on the `c89` utility.

Invoking a shell from TSO/E

To begin a z/OS UNIX shell session, you first log on to TSO/E and then invoke the TSO/E OMVS command. This starts a login shell, from which you can enter shell commands.

You can also login with `rlogin` or `telnet`.

See *z/OS UNIX System Services User's Guide* for additional information on starting a shell session.

Using the z/OS UNIX c89 utility to bind and create AMODE 64 executable files

To bind a z/OS UNIX C/C++ application's object modules to produce an AMODE 64 executable file, specify the `c89` utility and pass it object modules (*file.o* HFS files or *//file.OBJ* MVS data sets). The `c89` utility recognizes that these are object modules produced by previous C/C++ compilations and does not invoke the compiler for them.

Running under z/OS UNIX

To compile source files without binding them, use the `c89 -c` option to create object modules only. You can use the `-o` option with the command to specify the name and location of the executable file to be created.

For a complete description of all the `c89` options, see *z/OS UNIX System Services Command Reference*.

- To bind an AMODE 64 application object module to create the `mymod64.out` executable file in the current directory, specify:

```
c89_64 -o mymod64.out usersource.o
```

Invoking `c89_64` is specific to the `x1c` utility. The path must include the location where the `x1c` utility is installed because the `x1c` utility is not installed in the `/bin` directory.
- To bind an AMODE 64 application object module to create the default executable file `a.out` in the working directory, specify:

```
c89 -Wl,lp64 usersource.o
```
- To bind an AMODE 64 application object module to create the `mymod.out` executable file in the `app/bin` directory, relative to your working directory, specify:

```
c89 -o app/bin/mymod.out -Wl,lp64 usersource.o
```
- To bind several AMODE 64 application object modules to create the `mymod.out` executable file in the `app/bin` directory, relative to your working directory, specify:

```
c89 -o app/bin/mymod.out -Wl,lp64 usersrc.o ottrsrc.o "///PGM.OBJ(PW...APP)"
```
- To bind an AMODE 64 application object module to create the `MYLOADMD` executable member of the `MVS APPROG.LIB` data set for your user ID, specify:

```
c89 -o "///APPROG.LIB(MYLOADMD)" -Wl,lp64 usersource.o
```

Running z/OS UNIX AMODE 64 application programs using z/OS XL C/C++ functions

This section discusses the different ways you can run your z/OS UNIX AMODE 64 applications using z/OS XL C/C++ functions.

z/OS UNIX application program environments

z/OS UNIX supports the following environments from which you can run your z/OS UNIX applications using using z/OS XL C/C++ functions:

- z/OS UNIX shells
- TSO/E

You cannot directly call a z/OS UNIX application that resides in an HFS file from the TSO/E READY prompt. However, you can do so with a TSO/E `BPXBATCH` command, and with a `REXX EXEC`.

- MVS batch

You cannot directly use the `JCL EXEC` statement to run a z/OS UNIX application program that resides in an HFS file because you cannot put an HFS filename on the `JCL EXEC` statement. However, by using the `BPXBATCH` program, you can run a z/OS UNIX application that resides in an HFS file. You supply the name of the program as an argument to the `BPXBATCH` program, which runs under MVS batch and invokes a z/OS UNIX shell environment. (`BPXBATCH` also lets you call a program directly without having to also run a shell.) You can also run a z/OS UNIX application that resides in an HFS file by calling a `REXX EXEC` to invoke it under MVS batch.

Placing an MVS Application Executable Program in the File System

If you have a z/OS UNIX application executable file as a member in an MVS data set and want to place it in the HFS, you can use the OPUTX or OGETX z/OS UNIX TSO/E commands to copy the member into an HFS file. For a description of these commands, see *z/OS UNIX System Services Command Reference*. For examples of using these commands to copy data sets to HFS files, see *z/OS UNIX System Services User's Guide*.

Running an MVS Executable Program from a z/OS UNIX Shell

If your z/OS UNIX application resides in MVS data sets and you need to run the application executable program from within a shell, you can pass a call to the module to TSO/E. In many cases you can also use the tso utility. If you entered the shell from TSO/E using the OMVS command, you can use the TSO function key to pass the command to TSO/E. For example, if your executable program is myprog in data set my.loadlib, type the following (from the shell) to pass the command to TSO/E:

```
tso "call 'my.loadlib(myprog)'"
```

When the program completes, the shell session is restored. You can also run an MVS program from a shell by associating it with an HFS file by using sticky-bit or external link. See *z/OS UNIX System Services Command Reference* for more information on the chmod and the ln commands.

Running POSIX-enabled Programs

There are different considerations for running POSIX-enabled programs depending on whether you are using a z/OS UNIX shell or are running outside the shell.

Running POSIX-enabled Programs Using a z/OS UNIX Shell

Issuing the Executable from a Shell: Before an HFS program can be run in a shell, it must be given the appropriate mode authority for a user or group of users. You can update the mode authority for an executable by using the chmod command. See *z/OS UNIX System Services Command Reference* for the format and description of chmod. Note that when c89 creates an executable, the file is given execute permission for all users.

After you have updated the mode authority, enter the program name from the shell command line. For example,

- if you want to run the program data_crunch 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:

```
data_crunch
```

When running such programs, you can specify invocation runtime options only by setting the environment variable _CEE_RUNOPTS before invoking the program. For example, under a z/OS UNIX shell you can use the export command as shown below:

```
export _CEE_RUNOPTS="rpto(on)..."
```

To further update the runtime options, you can issue another export.

Running under z/OS UNIX

Issuing a Setup Shell Script from a Shell: To run a z/OS UNIX shell script that sets up an z/OS UNIX executable file and then runs the program, you give the appropriate mode authority for a user or group of users to run it. You can update the mode authority (access permission) for a shell script file by using the `chmod` command. See *z/OS UNIX System Services Command Reference* for the format and description of `chmod`. After mode authority has been given, enter the script file name from the shell command line.

Running POSIX-enabled Programs Outside the z/OS UNIX Shells

Running an MVS Batch z/OS UNIX Application File which is HFS-resident: To run a z/OS UNIX executable application file from an HFS file under MVS batch, invoke the IBM-supplied `BPX BATCH` program either from TSO/E, or by using JCL or a REXX EXEC (not batch). `BPX BATCH` performs an initial user login to run a specified program from the shell environment.

Before you invoke `BPX BATCH`, you must have the appropriate privilege to read from and write to HFS files. You should also allocate `STDOUT` and `STDERR` HFS files for writing any program output, such as error messages. Allocate the standard files using the `PATH` options on either the TSO/E `ALLOCATE` command or the JCL `DD` statement.

For a detailed discussion of the `BPX BATCH` program syntax and its use, and an example of running shell utilities under MVS batch using the `BPX BATCH` program, see *z/OS UNIX System Services Command Reference*.

Invoking BPXBATCH from TSO/E: You can invoke `BPX BATCH` from TSO/E in the following ways:

- From the TSO/E `READY` prompt
- From a `CALL` command
- As a REXX EXEC

If you want to run the `/myap/base_comp` application program from your user ID, direct its output to the file `/myap/std/my.out`. Write any error messages to the file `/myap/std/my.err` and copy the output and error data to MVS data sets. You could write a REXX EXEC similar to the following example:

```
/* base_comp REXX exec */
"Allocate File(STDOUT) Path('/u/myu/myap/std/my.out')
  Pathopts(OWRONLY,OCREAT,OTRUNC)
  Pathmode(SIRWXU) Pathdisp(DELETE,DELETE)"
"Allocate File(STDERR) Path('/u/myu/myap/std/my.err')
  Pathopts(OWRONLY,OCREAT,OTRUNC)
  Pathmode(SIRWXU) Pathdisp(DELETE,DELETE)"

"BPXBATCH PGM /u/myu/myap/base_comp"

"Allocate File(output1) Dataset('MYAPPS.STD(BASEOUT)')"
"Ocopy Indd(STDOUT) Outdd(output1) Text Pathopts(OVERRIDE)"

"Allocate File(output2) Dataset('MYAPPS.STD(BASEERR)')"
"Ocopy Indd(STDERR) Outdd(output2) Text Pathopts(OVERRIDE)"
```

Enter the name of the REXX EXEC from the TSO/E `READY` prompt to invoke `BPX BATCH`. When the REXX EXEC completes, the `STDOUT` and `STDERR` allocated files are deleted.

Invoking BPXBATCH Using JCL: To invoke `BPX BATCH` using JCL, submit a job that executes an application program and allocates the standard files using `DD` statements. For example, if you want to run the `/myap/base_comp` application

program from your user ID, direct its output to the file /myap/std/my.out. Direct any error messages to be written to the file /myap/std/my.err; code the JCL statements as follows:

```
//jobname JOB ...
//stepname EXEC PGM=BPXBATCH,PARM='PGM /u/myu/myap/base_comp'
//STDOUT DD PATH='/u/myu/myap/std/my.out',
//          PATHOPTS=OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
//STDERR DD PATH='/u/myu/myap/std/my.err',
//          PATHOPTS=OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
```

Invoking the Spawn Syscall in a REXX EXEC from TSO/E: A REXX EXEC can directly call a program which resides in the HFS. This can be done by using the spawn() syscall. The following is an example of a REXX program which can be called from TSO/E.

```
/* REXX */
RC = SYSCALLS('ON')
If RC<0 | RC>4 Then Exit RC
Address SYSCALL
fstdout = 'fstdout'
fstderr = 'fstderr'
'open' fstdout 0_RDWR+0_TRUNC+0_CREAT 700
stdout = RETVAL
'open' fstderr 0_RDWR+0_TRUNC+0_CREAT 700
stderr = RETVAL
map.0=-1
map.1=stdout
map.2=stderr
parm.0=1
parm.1='/bin/c89'
'spawn /bin/c89 3 map. parm. __environment.'
spid = RETVAL
serrno = ERRNO
If spid=-1 Then Do
  str ='unable to spawn' parm.1', errno='serrno
  'write' stderr 'str'
  Exit serrno
End
'waitpid (spid) waitpid. 0'
xrc = waitpid.W_EXITSTATUS
If xrc^=0 Then Do
  str =parm.1 'failed, exit status='xrc
  'write' stderr 'str'
End
Exit xrc
```

Running a z/OS UNIX Application Program Which is not HFS-resident: Submit a z/OS UNIX AMODE 64 application executable program using z/OS XL C/C++ functions (an executable file that is an MVS PDSE member) to run under the MVS batch environment using the JCL EXEC statement the same way you would submit a traditional C/C++ AMODE 31 application. The POSIX(ON) runtime option should be specified.

Chapter 6. Using runtime options

This chapter shows you how to specify runtime options as installation defaults, application defaults, in JCL, in commands, or in your source code. For more information about how Language Environment runtime options map to specific HLL options, see *z/OS Language Environment Programming Reference*.

Understanding the basics

Language Environment provides runtime options with which you can control certain aspects of your program's processing. You can set the default values for most of these options at installation time. See *z/OS Language Environment Programming Reference* for syntax and detailed information on the use of Language Environment runtime options.

Specifying runtime options

You can specify Language Environment runtime options in the following ways:

As installation defaults

The CELQDOPT assembler language source file establishes installation defaults for applications using the CEEXOPT macro. The file initially contains IBM-supplied default values for each of the Language Environment runtime options. During installation of Language Environment, the default values contained in the CELQDOPT source file can be edited and assembled to create the CELQDOPT object module.

It is possible to associate a nonoverrideable attribute with each individual runtime option. Each option in CELQDOPT must be specified as either overrideable (OVR) or nonoverrideable (NONOVR). This allows the installation to enforce options that are critical to the overall Language Environment operating environment. All applications that run in the common runtime environment operate using these default values for the runtime options.

The syntax for the CEEXOPT macro is presented in the section, "CEEXOPT invocation syntax" on page 51. See *z/OS Language Environment Customization* for more information on modifying the installation default runtime options.

As system-level defaults

Language Environment, along with z/OS, also provides another level of runtime options through a member in the system parmlib. The format of the parmlib member name is CEEPRMxx. The member is identified during IPL by a CEE=xx statement, either in the IEASYSy data set or in the IPL PARMS. After IPL, the active parmlib member can be changed with the SET CEE=xx command. Individual options may be changed using the SETCEE command.

It is possible to associate a nonoverrideable attribute with each individual run-time option. Each option can be specified as overrideable (OVR) or nonoverrideable (NONOVR). This allows the system to enforce options that are critical to the overall Language Environment operating environment.

Using runtime options

For more information on specifying options at system initialization, see *z/OS MVS Initialization and Tuning Reference*.

As region-wide defaults

The CELQROPT assembler language source program establishes region-wide defaults using the CEEXOPT macro for a particular region. CELQROPT is optional. If it is used, you should code just the runtime options to be changed from the installation or system-level defaults. Runtime options which are omitted from CELQROPT will remain the same as the installation defaults and system-level defaults. The CELQROPT module resides in a user-specified load library.

It is possible to associate a nonoverrideable attribute with each individual runtime option. Each option in CELQROPT must be specified as either overrideable (OVR) or nonoverrideable (NONOVR). This allows the region to enforce options that are critical to the operation of the region. CELQROPT should not specify OVR for a runtime option which is defined as NONOVR in the installation defaults, as this will be ignored and a warning message will be issued.

The CEEPRMxx parmlib member keyword CELQROPT controls if region-wide defaults are allowed.

For more information, see *z/OS Language Environment Customization*.

As application defaults

The CELQUOPT assembler language source program establishes application defaults using the CEEXOPT macro. CELQUOPT can be edited and assembled to create an object module, CELQUOPT, that can be linked with an application. As noted previously, CELQDOPT establishes installation defaults using the CEEXOPT macro. When the application runs, the options specified in CELQUOPT override any corresponding overrideable CELQDOPT options.

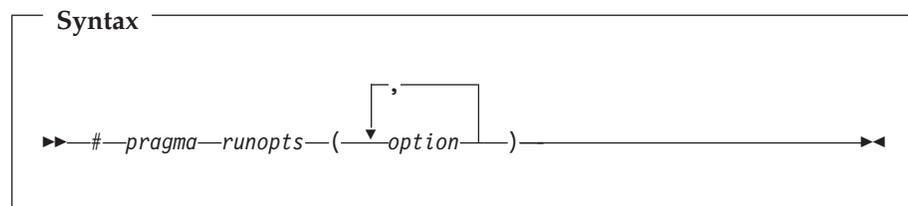
CELQUOPT must be link-edited with your application to establish application defaults.

In your C/C++ source code:

C provides the `#pragma runopts` directive, with which you can specify runtime options in your source code.

You must specify `#pragma runopts` in the source file that contains your main function, before the first C statement. Only comments and other pragmas can precede `#pragma runopts`.

Specify `#pragma runopts` as follows:



option is a Language Environment runtime option.

Restriction:

For C++ applications, the following values are not allowed for compilation:

- NOEXECOPS | EXECOPS
- NOREDIR | REDIR
- NOARGPARSE | ARGPARSE

Instead, you must use the corresponding C++ compiler options.

For more information about using C/C++ pragmas, see *z/OS XL C/C++ User's Guide*.

In JCL

You can specify runtime options in the PARM parameter of the JCL EXEC statement or as a DD card named CEEOPTS. See “Specifying runtime options in the EXEC statement” on page 33 and “Specifying Run-Time Options with the CEEOPTS DD card” on page 34 for details.

In TSO/E commands, on application invocation

You can specify runtime options as options on the CALL command. See *z/OS Language Environment Programming Guide* for more information.

In C/C++ applications invoked by one of the exec or spawn family of functions

You can specify the values of runtime options by using the `_CEE_RUNOPTS` environment variable. For more information, see “Using `_CEE_RUNOPTS` to specify runtime options” on page 50.

Order of precedence

It is possible for more than one of the methods listed above to be used for a given application. The order of precedence (from highest to lowest) between option specification methods is:

1. Options specified on invocation of the application (or in the case of an application invoked by one of the exec or spawn family of functions, options specified in the environment variable `_CEE_RUNOPTS`).
2. Options specified at invocation time through a DD card (DD:CEEOPTS). The CEEOPTS DD is ignored for programs invoked using one of the `exec()` family of functions.
3. Options specified in a CELQUOPT CSECT. There are a few methods available to provide a CELQUOPT CSECT:
 - Assemble a CELQUOPT. For more information, see *z/OS Language Environment Customization*.
 - Specify the C/C++ `#pragma runopts()` directive within a source program. The compiler generates the CELQUOPT CSECT from the given options.

If you select `#pragma runopts()`, specify it in one and only one compile unit in the application; for example, in the main routine. If multiple CELQUOPTs are present, binder input ordering determines which CELQUOPT is used in an executable program. Only the first CELQUOPT CSECT linked in an executable program is applied. The binder treats any subsequent CELQUOPTs seen in the input as duplicates and they will be ignored.

4. Region-wide default options defined within CELQROPT.
5. Options changed after IPL with the SET CEE=xx or SETCEE command. The member will be parsed and the options table will be replaced with the contents of the new members.

Using runtime options

- Options specified during IPL by a CEE=(xx,yy,...) statement, either in the IEASYSyy data set or in the IPL PARMS.
- Option defaults defined at installation time.

When the non-overrideable (NONOVR) attribute is specified for a run-time option, all methods of specifying that run-time option with higher precedence are ignored.

Specifying suboptions in runtime options

Use commas to separate suboptions of runtime options. If you do not specify a suboption, you must still specify the comma to indicate its omission, for example STACK64(, ,512M). However, trailing commas are not required; STACK64(1M,2M) is valid. If you do not specify any suboptions, either of the following is valid: STACK64 or STACK64().

Specifying runtime options and program arguments

To distinguish runtime options from program arguments that are passed to Language Environment, the options and program arguments are separated by a forward slash (/). For more information on program arguments, see “Argument lists and parameter lists” on page 59.

Runtime options precede program arguments whenever they are specified in JCL or in TSO/E commands on application invocation.

Table 7. Formats for specifying runtime options and program arguments

When...	Format
Only runtime options are present	runtime options/
Only program arguments are present	One of the following:
1. If a slash is present in the arguments, a preceding slash is mandatory.	1. /program arguments
2. If a slash is not present in the arguments, a preceding slash is optional.	2. program arguments <i>or</i> /program arguments
Both runtime options and program arguments are present	runtime options/program arguments

Using `_CEE_RUNOPTS` to specify runtime options

If you run C/C++ applications that are invoked by one of the exec or spawn family of functions, you can use the environment variable `_CEE_RUNOPTS` to specify invocation Language Environment runtime options. For more information on using the environment variable `_CEE_RUNOPTS`, see *z/OS XL C/C++ Programming Guide*.

C and C++ compatibility considerations

C provides the `#pragma runopts` directive for you to specify runtime options in your source code. When `#pragma runopts(execops)` is in effect (the default), you can pass runtime options from the command line. Runtime options must be followed by a slash (/).

If the main routine is C and `#pragma runopts(noexecops)` is specified in the source, you cannot enter runtime options on the command line. Language Environment interprets the entire string on the command line including runtime options, if present, as program arguments to the main routine.

For a description of the EXECOPS runtime option, see *z/OS Language Environment Programming Reference*.

CEEXOPT invocation syntax

Use the CEEXOPT macro to establish installation, region-wide, and programmer default options.

This macro definition can be found in the Language Environment SCEEMAC data set, which must be specified on the SYSLIB DD statement when assembling the options.

- When invoked during the assembly of the CELQDOPT source program at installation time, CEEXOPT creates the CELQDOPT object module, which establishes installation default options. All Language Environment runtime options must be specified in CELQDOPT. Each option in CELQDOPT must be designated as either overrideable (OVR) or nonoverrideable (NONOVR). In addition, a valid value must be specified for each suboption of each runtime option.
- The CEEXOPT macro also creates the CELQROPT object module when CELQROPT is assembled. CELQROPT can then be linked as a stand-alone module to establish region-wide runtime options. Since it is not link-edited with any other Language Environment modules, Language Environment installation and system-level runtime option defaults can be changed at any time on a region-by-region basis. While the use of CELQROPT itself is optional, and any runtime options in it are also optional, any runtime option that is specified must be designated as either overrideable (OVR) or nonoverrideable (NONOVR). In addition, a valid value must be specified for each suboption of each specified runtime option.
- The CEEXOPT macro also creates the CELQUOPT object module when CELQUOPT is assembled. CELQUOPT can be linked with an application program to establish application default options. Options in CELQUOPT's invocation of CEEXOPT cannot be designated as overrideable or nonoverrideable. However, their suboption values take precedence over those of any corresponding overrideable CELQDOPT or CELQROPT option values. For information on CELQDOPT or CELQROPT, see *z/OS Language Environment Customization*.

To invoke CEEXOPT to create the CELQUOPT module to specify user runtime option defaults, use the format of the IBM-supplied template CELQUOPT, as shown in Figure 15 on page 52. It is a sample of CELQUOPT with the default suboption values for each of the runtime options, which establish application defaults.

```

CELQUOPT CSECT
CELQUOPT AMODE 64
CELQUOPT RMODE ANY
    CEEXOPT CEEDUMP=(60,SYSOUT=*,FREE=END,SPIN=UNALLOC)           X
        DYNDUMP=(*USERID,NODYNAMIC,TDUMP),                       X
        ENVAR=(' '),                                             X
        FILETAG=(NOAUTOCVT,NOAUTOTAG),                             X
        HEAPCHK=(OFF,1,0,0,0,1024,0,1024,0),                       X
        HEAPPOLS=(OFF,8,10,32,10,128,10,256,10,1024,10,          X
        2048,10,0,10,0,10,0,10,0,10,0,10),                       X
        HEAPPOLS64=(OFF,8,4000,32,2000,128,700,256,350,          X
        1024,100,2048,50,3072,50,4096,50,8192,25,16384,10,      X
        32768,5,65536,5),                                         X
        HEAP64=(1M,1M,KEEP,32K,32K,KEEP,4K,4K,FREE),            X
        INFOMSGFILTER=(OFF,,,),                                   X
        IOHEAP64=(1M,1M,FREE,12K,8K,FREE,4K,4K,FREE),           X
        LIBHEAP64=(1M,1M,FREE,16K,8K,FREE,8K,4K,FREE),          X
        NATLANG=(ENU),                                           X
        NOTEST=(ALL,*,PROMPT,INSPREF),                             X
        POSIX=(OFF),                                             X
        PROFILE=(OFF,' '),                                       X
        RPTOPTS=(OFF),                                           X
        RPTSTG=(OFF),                                           X
        STACK64=(1M,1M,128M),                                    X
        STORAGE=(NONE,NONE,NONE,),                               X
        THREADSTACK64=(OFF,1M,1M,128M),                          X
        TERMTHDACT=(TRACE,,96),                                  X
        TRACE=(OFF,,DUMP,LE=0),                                   X
        TRAP=(ON,SPIE)                                           X
    END

```

Figure 15. Sample invocation of CEEXOPT within CELQUOPT source program

Notes on CEEXOPT invocation

You should be aware of the following considerations when invoking CEEXOPT:

- A continuation character (X in the source) must be present in column 72 on each line of the CEEXOPT invocation, except the last line. This applies to CELQUOPT, CELQROPT, and CELQDOPT.
- Options and suboptions must be specified in uppercase. Only suboptions that are strings can be specified in mixed case or lowercase.
- A comma must end each option except for the final option. If the comma is omitted, everything following the option is treated as a comment.
- If one of the string suboptions contains a special character, such as embedded blank or unmatched right or left parenthesis, the string must be enclosed in apostrophes (' '), not in quotation marks (" "). A null string can be specified with either adjacent apostrophes or adjacent quotation marks.
 To get a single apostrophe (') or a single ampersand (&) within a string, two instances of the character must be specified. The pair is counted as only one character in determining if the maximum allowable string length has been exceeded, and in setting the effective length of the string.
- Macro instruction operands cannot be longer than 1024 characters. If the number of characters to the right of the equal sign is greater than 1024 for any keyword parameter in the CEEXOPT invocation, a return code of 12 is produced for the assembly, and the options are not parsed properly.
- Avoid unmatched apostrophes in any string. The error cannot be captured within CEEXOPT itself; instead, the assembler produces a message such as

```
IEV063 *** ERROR *** NO ENDING APOSTROPHE
```

which bears no particular relationship to the suboption in which the apostrophe was omitted. Furthermore, none of the options is properly parsed if this mistake is made.

- You can completely omit the specification of any runtime option in CELQUOPT or CELQROPT.
 - In CELQROPT, a valid value must be specified for each suboption of those runtime options that are included.
 - In CELQUOPT you can completely omit the specification of any suboption of those runtime options which are included. Default values are then supplied for each of the missing suboptions in the options control block that is generated, and these values are ignored at the time Language Environment merges the options.

There are two recommended ways of omitting a suboption in CELQUOPT. The HEAP64 runtime is used below to demonstrate:

- Specify the option with only a comma following the equal sign:

```
HEAP64=, X
```

or

- Specify the option with empty parentheses and comma following the equal sign:

```
HEAP64=(), X
```

In either case, the continuation character (X in this example) must still be present in column 72.

- In CELQUOPT you can use commas to indicate the omission of one or more suboptions for options having more than one suboption. For example, if you wish to specify only the second suboption of the STORAGE option, the omission of the 1st, 3rd, and 4th suboptions can be indicated in any of the following ways:

```
STORAGE=(,NONE), X
STORAGE=(,NONE,), X
STORAGE=(,NONE,,), X
```

Because suboptions are positional parameters, do not omit the comma if the corresponding suboption is omitted and another suboption follows.

- Options that permit only one suboption do not need to enclose that suboption in parentheses. For example, the NATLANG option can be specified in CELQUOPT in either of the following ways:

```
NATLANG=(ENU), X
NATLANG=ENU, X
```

Performance considerations

For optimal performance when using CELQUOPT, code only those options that you want to change. This enhances performance by minimizing the number of options lines Language Environment must scan. Options and suboptions that are to remain the same as the installation defaults do not need to be repeated. For example, if the only change you want to make is to define STACK64 with an initial value of 2M and an increment of 2M, include only that runtime option in CELQUOPT as shown in the following example:

Using runtime options

```

CELQOPT CSECT
CELQOPT AMODE 64
CELQOPT RMODE ANY
CEEEOPT STACK64=(2M,2M)
END

```

Parmlib Syntax

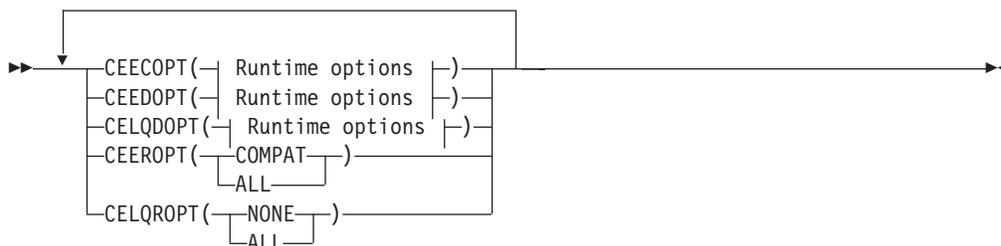
z/OS provides parmlib members for specifying defaults for many system options. The CEEPRMxx member can be used for specifying system-level default run-time options that control various aspects of Language Environment. CEEPRMxx is identified during IPL by a CEE=xx statement, either in the IEASYSyy parmlib member or in the response to the System Parameters prompt at IPL. After IPL, the operator can do the following:

- Change the member with the SET CEE=xx command.
- Change individual options using the SETCEE command.
- Display current option settings with the D CEE command.
- Clear all the system-level default run-time options and keywords using the SETCEE CLEAR command.

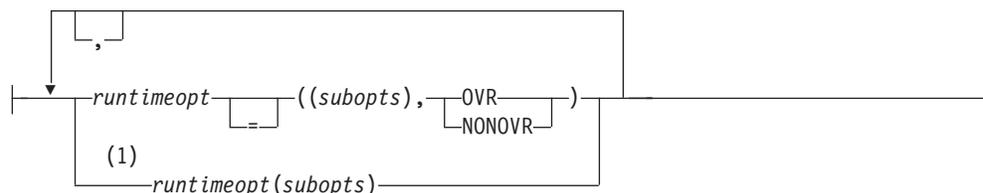
Each run-time option can be specified as overrideable (OVR) or non-overrideable (NONOVR). This allows the system to enforce options that are critical to the overall Language Environment operating environment.

Guideline: Using this support is not required, so the default IEASYS00 member does not specify a CEEPRMxx member. If you want to use this support, a CEEPRM00 member is included in SCEESAMP.

The format of the CEEPRMxx member is:



Runtime options:



Notes:

- 1 Specifying run-time options without the OVR or NONOVR attribute. Run-time options specified using this format can be overridden.

Suggestion: Use the CEEPRMxx support to set system-level defaults for Language Environment instead of using the CEEWCOPT, CEEWDOPT and CEEWQDOP sample jobs.

For more information about the CEEPRMxx PARMLIB member, see *z/OS Language Environment Customization*.

CEEOPTS DD Syntax

To specify the CEEOPTS DD statement, use the following syntax:

- For in-stream JCL:

```
//CEEOPTS DD *  
POSIX(ON),STACK(2M,2M,256M)
```

- For a sequential data set:

```
//CEEOPTS DD DSN=MY.CEEOPTS.DATASET,DISP=SHR
```

- For a partitioned data set:

```
//CEEOPTS DD DSN=MY.CEEOPTS.DATASET(MYOPTS),  
// DISP=SHR
```

- To ignore the DD statement:

```
//CEEOPTS DD DUMMY
```

For more information, see section "Using the CEEOPTS DD statement" in *z/OS Language Environment Customization*.

Part 2. Preparing an application to run with Language Environment

Chapter 7. Using Language Environment parameter list formats

This chapter describes how to pass parameters to external routines under Language Environment. The methods described do not apply to internal routines or to compiled code that invokes its own library routines.

Understanding the basics

When writing a Language Environment-conforming AMODE 64 application, it is important to consider how parameters are passed to the application on invocation. The type of parameter list created by the operating system and passed to Language Environment when an application is run varies according to the operating system or subsystem used. Language Environment repackages the various formats so that what is actually passed to the main routine when it is invoked on most supported operating systems is a halfword prefixed character string. In C and C++, you can pass arguments to the main routine through `argv` and `argc`. If you set up your C or C++ main routine according to the rules of the language, you generally do not need to do anything special to receive parameters from the operating system. In this case, use the constructs provided by the C or C++ language. Refer to *z/OS XL C/C++ Language Reference* for more details.

Argument lists and parameter lists

Figure 16 summarizes the terminology used with Language Environment to describe passing parameters to and from routines. In Figure 16, a calling routine passes an *argument list* to a called routine. That same list is referred to as a *parameter list* when it is received by the called routine. Under Language Environment, the formats of the argument and parameter lists are identical. The only difference between the two terms is whether they are being used from the point of view of the calling or the called routine.

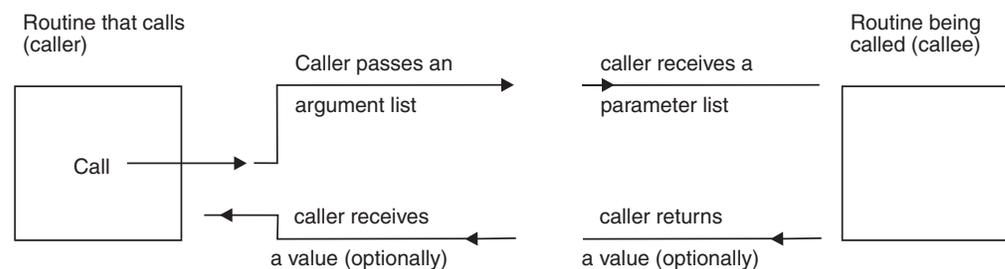


Figure 16. Call Terminology Refresher

Passing arguments between routines

Language Environment-conforming HLLs use the semantic terms *by value* and *by reference* to indicate how changes in the argument values for a called routine affect the calling routine:

By value

Any changes made to the argument value by the called routine will not alter the original argument passed by the calling routine.

Language Environment Parameter List Formats

By reference

Changes made by the called routine to the argument value can alter the original argument value passed by the calling routine.

Under Language Environment you can pass arguments directly and indirectly as follows:

Direct The value of the argument is passed directly in the parameter list. You cannot pass an argument by reference (direct).

Indirect

A pointer to the argument value is passed in the parameter list.

Table 8 summarizes the semantic terms by value and by reference and the direct and indirect methods for passing arguments. The table shows what is passed to routines.

Table 8. Semantic Terms and Methods for Passing Arguments in Language Environment

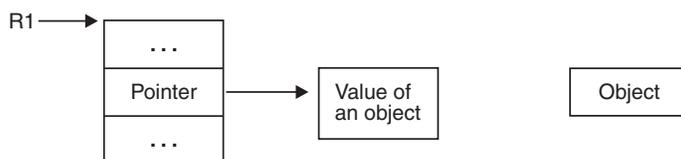
	By Value	By Reference
Direct	The value of the object is passed	Not allowed under Language Environment
Indirect	A pointer points to the value of an object	A pointer points to the object

Figure 17 illustrates these argument passing styles. In Figure 17, register 1 (R1) points to the value of an object, or to an argument list containing either a pointer to the value of an object or a pointer to the object.

By Value (Direct)



By Value (Indirect)



By Reference (Indirect)



Figure 17. Argument Passing Styles in Language Environment

HLL semantics usually determine when data is passed by value or by reference. The AMODE 64 support provided by Language Environment supports argument passing styles as shown in Table 9 on page 61.

Table 9. Default Passing Style per HLL

Language	Default Argument
C	By Value (Direct)
C++	By Value (Direct) ¹

Notes:

1. C++ also supports by reference (indirect), if a prototype specifies it (with ampersand (&)).

Language Environment Parameter List Formats

Chapter 8. Making Your Application Reentrant

This chapter shows you how to make your AMODE 64 application reentrant. *Reentrancy* allows more than one user to share a single copy of a load module. If your application is not reentrant, each application that calls your application must load a separate copy of your application.

Understanding the Basics

The following routines **must** be reentrant:

- Routines to be loaded into the LPA or ELPA

Your routine should be reentrant if it is a large routine that is likely to have multiple concurrent users. Less storage is used if multiple users share the routine concurrently. Reentrancy also offers some performance enhancement because there is less paging to auxiliary storage.

If you want your routine to be reentrant, ensure that it does not alter any static storage that is part of the executable program; if the static storage is altered, the routine is not reentrant and its results are unpredictable.

Making Your C/C++ Program Reentrant

Under C/C++, reentrant programs can be categorized by their reentrancy type as follows:

Natural reentrancy

The attribute of programs that contain no modifiable external data.

Natural reentrancy is not applicable to C++.

Constructed reentrancy

The attribute of applications that contain modifiable external data and require additional processing to become reentrant. By default, all C++ programs are made reentrant via constructed reentrancy.

Natural Reentrancy

A C program is naturally reentrant if it contains no modifiable external data. In C, the following are considered modifiable external data:

- Variables using the `extern` storage class
- Variables using the `static` storage class
- Writable strings

If your C program is naturally reentrant, you do not need to use the `RENT` compiler option. After compiling and binding, install it in one of the locations listed in “Installing a Reentrant Load Module” on page 64.

Constructed Reentrancy

A constructed reentrant program is created by using the binder to combine all of the object modules produced by the XL C/C++ compiler. The target data set for the AMODE 64 executable must be a PDSE or the UNIX file system.

Reentrant Applications

The compile-time initialization information from one or more object modules is combined into a single initialization unit.

Programs with constructed reentrancy are split into two parts:

- A variable or nonreentrant part that contains external data
- A constant or reentrant part that contains executable code and constant data

Each user running the program receives a private copy of the first part (mapped by the binder), which is initialized at run time. The second part can be shared across multiple spaces or sessions only if it is installed with dynamic LPA.

Generating a Reentrant Program Executable for C or C++

To generate a reentrant C object module, follow these steps:

1. For C, if your program contains external data, compile your source files using the RENT (and LP64) compiler option. For C++, compile your source files with LP64; by default the compiler builds reentrant programs using constructed reentrancy. See *z/OS XL C/C++ User's Guide* for more information.
2. To produce an executable program, use the binder to combine all of the input into an AMODE 64 executable program.
3. To get the greatest benefit from reentrancy, install your executable program in one of the locations listed in "Installing a Reentrant Load Module."

Installing a Reentrant Load Module

You will get the most benefit from reentrancy if you link the program with the RENT attribute and any other attributes you would normally use, and have your system programmer install the load module in the link pack area of the system using dynamic LPA.

Modules may be added/removed from the dynamic LPA after an IPL using the SET PROG=MVS console command.

Part 3. Language Environment concepts, services, and models

This section provides more information about Language Environment and the services it provides.

Chapter 9. Initialization and termination under Language Environment

This chapter describes initialization and termination of AMODE 64 applications under Language Environment. It describes how you can customize your AMODE 64 applications during initialization and termination by using Language Environment runtime options and APIs.

Understanding the basics

Initialization and termination establish the state of various parts of the Language Environment program model. The program model describes three major entities of a program structure:

Process	A collection of resources (code and data).
Enclave	A collection of program units consisting of exactly one main routine and zero or more subroutines.
Thread	The basic unit of execution.

The z/OS UNIX System Services (z/OS UNIX) program model differs somewhat from the Language Environment program model. Refer to “Mapping the POSIX program model to the Language Environment program model” on page 77 for more information. For more detailed definitions of program model and other Language Environment terms, see Chapter 10, “Program model,” on page 73.

When you run a routine, Language Environment initializes the runtime environment by creating a process, an enclave, and an initial thread.

During termination, all threads, the enclave and process are terminated.

Language Environment initialization

During initialization, a process, an enclave, and then an initial thread are created.

Process initialization sets up the framework to manage an enclave. Enclave initialization creates the framework to manage enclave-related resources and the threads that run within the enclave. Thread initialization acquires a stack and enables the condition manager for the thread.

Language Environment termination

Language Environment termination provides services that restore the operating environment to its original state after your application either runs to completion or terminates abnormally. You can affect termination through the use of runtime options and APIs.

What causes termination

Under Language Environment, an application terminates when any of the following conditions occur:

- The last thread in the enclave terminates (which in turn terminates the enclave).
- The main routine in the enclave returns to its caller.

Initialization and Termination

- An HLL construct issues a request for the termination of an enclave, for example:
 - C's `abort()` function
 - C's `raise(SIGTERM)` function
 - C's `_exit()` function
 - C's `exit()` function
- A default POSIX signal is received, where the default is termination.
- An abend is requested by the application (that is, the application calls `__cabend()`).
- An unhandled condition of severity 2 or greater occurs. (See “Termination behavior for unhandled conditions” on page 71 for information.)

What happens during termination

The following sequence of events occurs during termination:

1. C `atexit()` functions and c++ static destructors are invoked, if present. They are not invoked if `_exit` calls for termination or if abnormal termination occurs.
2. For normal termination, the enclave return code is set (see “Managing return codes in Language Environment” on page 69). For abnormal termination caused by an unhandled condition of severity 2 or greater, an abend is returned (see “Termination behavior for unhandled conditions” on page 71).
3. The environment is terminated:
 - The enclave is terminated
 - All enclave resources are returned to the operating system
 - Any files that Language Environment manages are closed
 - The debugger is terminated, if active
 - The profiler is terminated, if active

When a condition of severity 2 and/or greater occurs, depending on the setting of the `TERMTHDACT` runtime option, you might receive a message, a trace of the active routines, or a dump. For more information on `TERMTHDACT`, see *z/OS Language Environment Programming Reference*.

Thread termination

A thread terminating in a non-POSIX environment is analogous to an enclave terminating, because Language Environment supports only single threads. See “Enclave termination” on page 69 for information on enclave termination.

POSIX thread termination: A thread terminates due to `pthread_exit()`, `pthread_kill()`, or `pthread_cancel()`, or simply returns from the start routine of the thread in a POSIX environment. When a thread issues a `exit()` or `_exit()` or encounters an unhandled condition, that thread terminates and all other active threads are also forced to terminate. The z/OS UNIX (POSIX) environment supports multiple threads; each thread is terminated, as follows:

- The stack storage associated with the thread is freed
- The thread status is set
- Cleanup handlers and destructor routines are driven
- The stack is collapsed

For more detailed information on POSIX functions, refer to the following resources:

- “Language Environment and POSIX Signal Handling Interactions” on page 106
- “Mapping the POSIX program model to the Language Environment program model” on page 77
- *z/OS UNIX System Services User's Guide*

Enclave termination

When an enclave terminates, Language Environment releases resources allocated on behalf of the enclave and performs various other activities including the following:

- If present, calls `atexit()` functions and C/C++ static destructors
- Calls HLL-specific termination routines for HLLs that were active during the executing of the program
- Deletes modules loaded by Language Environment
- Frees all storage obtained by Language Environment services
- Frees Language Environment control blocks for the enclave
- Language Environment sets a return code and reason code or an `abend`
- Restores the program mask and registers to preinitialization values
- Returns control to the enclave creator

Process termination

Process termination occurs when the enclave terminates. Process termination releases the process control block (PCB) and associated resources, and returns control to the creator of the process.

Language Environment explicitly relinquishes all resources it acquires. Routines that acquire resources directly from the host system (such as opening a DCB) must explicitly relinquish the resource. If these resources are not explicitly released, the environment can be corrupted because Language Environment has no method for releasing these resources.

POSIX process termination: In a z/OS UNIX environment, POSIX process termination maps to Language Environment enclave termination. For specific information on POSIX default signal action at POSIX process termination when running in an z/OS UNIX environment, see “Language Environment and POSIX Signal Handling Interactions” on page 106.

In a z/OS UNIX environment, the following occurs if the process being terminated is a child process:

- The parent process is notified with a `wait` or a `waitpid` or saving of the exit status code.
- A new parent process ID is assigned to all child processes of the terminated process.
- If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session allowing it to be acquired by a new controlling process.

Managing return codes in Language Environment

This section discusses how Language Environment calculates and uses return codes and reason codes during enclave termination. The return codes between subroutine calls that are implemented with programming language constructs are addressed in the appropriate language-specific programming guides.

How the Language Environment enclave return code is calculated

When an enclave terminates, Language Environment provides a Language Environment enclave return code and an enclave reason code (sometimes called a

Initialization and Termination

return code modifier). The Language Environment enclave return code is calculated by summing the user return code generated by the HLL and the enclave reason code as follows:

Language Environment enclave return code = user return code + enclave reason code

The Language Environment enclave return code is placed in register 15, and the enclave reason code is placed in register 0.

Setting and altering user return codes

User return codes can be set and altered by language constructs. As described in the following sections, the user return code value is based on the reason an enclave terminates and the language of the routine that initiates termination.

For C and C++

If a normal return from `main()` terminates the application, the user return code value is 0. When a C or C++ routine terminates an enclave with a language construct such as `exit(n)` or `return(n)`, the value of *n* is used.

If the enclave terminates due to an unhandled condition of severity 2 or greater, the user return code value is 0. For information on unhandled conditions, see "Termination behavior for unhandled conditions" on page 71. For more information about C or C++ language constructs, see *z/OS XL C/C++ Programming Guide*.

How the enclave reason code is calculated

The enclave reason code provides additional information in support of the enclave return code. Language Environment calculates the enclave reason code by multiplying a severity code (that indicates how an enclave terminated) by 1000.

The severity code is initially set to 0, indicating normal enclave termination. If the Termination_Imminent due to STOP (T_I_S) condition is signaled, it is set to 1. If the enclave terminates due to an unhandled condition of severity 2 or greater, the enclave reason code is set according to the severity of the unhandled condition that caused the enclave to terminate, as shown in Table 10. For more information about Language Environment conditions and severity codes, see Table 14 on page 95.

Table 10 contains a summary of the enclave reason code produced when an enclave terminates. The condition severity column indicates the reason code for the original condition.

Table 10. Summary of Enclave Reason Codes

Condition Severity	Meaning	Enclave Reason Code — (R0)
0	Normal application termination	0
Severity 1 condition	Termination_Imminent due to STOP	1000
Unhandled severity 2 condition	Error — abnormal termination	abend
Unhandled severity 3 condition	Severe error — abnormal termination	abend
Unhandled severity 4 condition	Critical error — abnormal termination	abend

Termination behavior for unhandled conditions

When there is an unhandled condition of severity 2 or greater, an enclave terminates with an abend. Language Environment will assign an abend code, return code and reason code, as described in this section. You can also assign values yourself, as described in “Setting and altering user return codes” on page 70.

For a discussion of conditions and how they are handled in Language Environment, see Table 14 on page 95. For specific information pertaining to POSIX signal action defaults and unhandled conditions in a z/OS UNIX environment, see “Language Environment and POSIX Signal Handling Interactions” on page 106.

Determining the abend code

Language Environment terminates the enclave with the same abend code that caused the unhandled condition of severity 2 or greater if the unhandled condition was generated by an abend.

Table 11 shows the abend code and reason code used when the enclave terminates due to the various unhandled conditions of severity 2 or greater.

Table 11. Abend Code Values Used by Language Environment

Unhandled Condition	Abend Code	Abend Reason Code
ABEND	The original abend code	The original abend reason code
Program interrupt	For program interrupt abend codes, see “Program interrupt abend and reason codes”	
Software-raised condition	A user 4038 abend	X'1'
Unsuccessful LOAD	The abend code that would have been used by the operating system.	The abend reason code that would have been used by the operating system.

Program interrupt abend and reason codes

A program interrupt can cause an unhandled condition of severity 2 or greater. The abend codes and reason codes shown in Table 12 are issued for program interrupts.

Table 12. Program Interrupt Abend and Reason Codes

Program Interrupts	Abend Code	Abend Reason Code
Operation exception	S0C1	00000001
Privileged operation exception	S0C2	00000002
Execute exception	S0C3	00000003
Protection exception	S0C4	00000004
Segment translation exception (note 1)	S0C4	00000004
Page translation exception (note 2)	S0C4	00000004
Addressing exception	S0C5	00000005
Specification exception	S0C6	00000006
Data exception	S0C7	00000007
Fixed-point overflow exception	S0C8	00000008

Initialization and Termination

Table 12. Program Interrupt Abend and Reason Codes (continued)

Program Interrupts	Abend Code	Abend Reason Code
Fixed-point divide exception	S0C9	00000009
Decimal overflow exception	S0CA	0000000A
Decimal divide exception	S0CB	0000000B
Exponent overflow exception	S0CC	0000000C
Exponent underflow exception	S0CD	0000000D
Significance exception	S0CE	0000000E
Floating-point divide exception	S0CF	0000000F

Notes:

1. The operating system issues abend code S0C4 reason code 10 for segment translation program interrupts.
2. The operating system issues abend code S0C4 reason code 11 for page translation program interrupts.

Chapter 10. Program model

Now that you have been introduced to how AMODE 64 applications run in Language Environment, you need to understand the model of program management under which Language Environment operates. This chapter provides an overview of the Language Environment model.

The Language Environment program model supports the language semantics of applications that run in the common run-time environment and defines the way routines or programs are put together to form an application. Language Environment implements a subset of the POSIX program model. Features not supported in z/OS Language Environment are indicated in this manual.

The POSIX program model differs somewhat from the Language Environment program model. Refer to “Mapping the POSIX program model to the Language Environment program model” on page 77 for more information.

Understanding the basics

The Language Environment program model has three basic entities — the process, enclave, and thread, each of which Language Environment creates whenever you start execution of an AMODE 64 application. This section describes each of these entities and their relationship to program management.

Language Environment program model terminology

Some terms used to describe the program model are common programming terms; others have meanings that are specific to a given language. It is important that you understand the meaning of the terminology Language Environment uses and how it compares with existing languages. For more detailed definitions of these and other Language Environment terms, please consult the glossary in *z/OS Language Environment Concepts Guide*.

Language Environment terms and their HLL equivalents

Process	The highest level of the Language Environment program model; a collection of resources, both program code and data, consisting of at least one enclave.
Enclave	The enclave defines the scope of HLL semantics. In Language Environment, a collection of routines, one of which is designated as the main routine. The enclave contains at least one thread.
Thread	An execution entity that consists of synchronous invocations and terminations of routines. The thread is the basic run-time path within the Language Environment program model; dispatched by the system with its own run-time stack, instruction counter, and registers.
Routine	In Language Environment, either a procedure, function, or subroutine.

Terminology for data

Automatic data

Data that does not persist across calls. In the absence of a specific

Program model

initializer, automatic data get “accidental” values that may depend on the behavior of the caller or the last function to be called by the caller.

External data

Data with one or more named points by which the data can be referenced by other program units and data areas. External data is known throughout an enclave.

Local data

Data known only to the routine in which it is declared; equivalent to local data in C, C++.

Figure 18 shows the simplest form of the Language Environment program model and the resources that each component controls. Refer to the figure as you read about the program model.

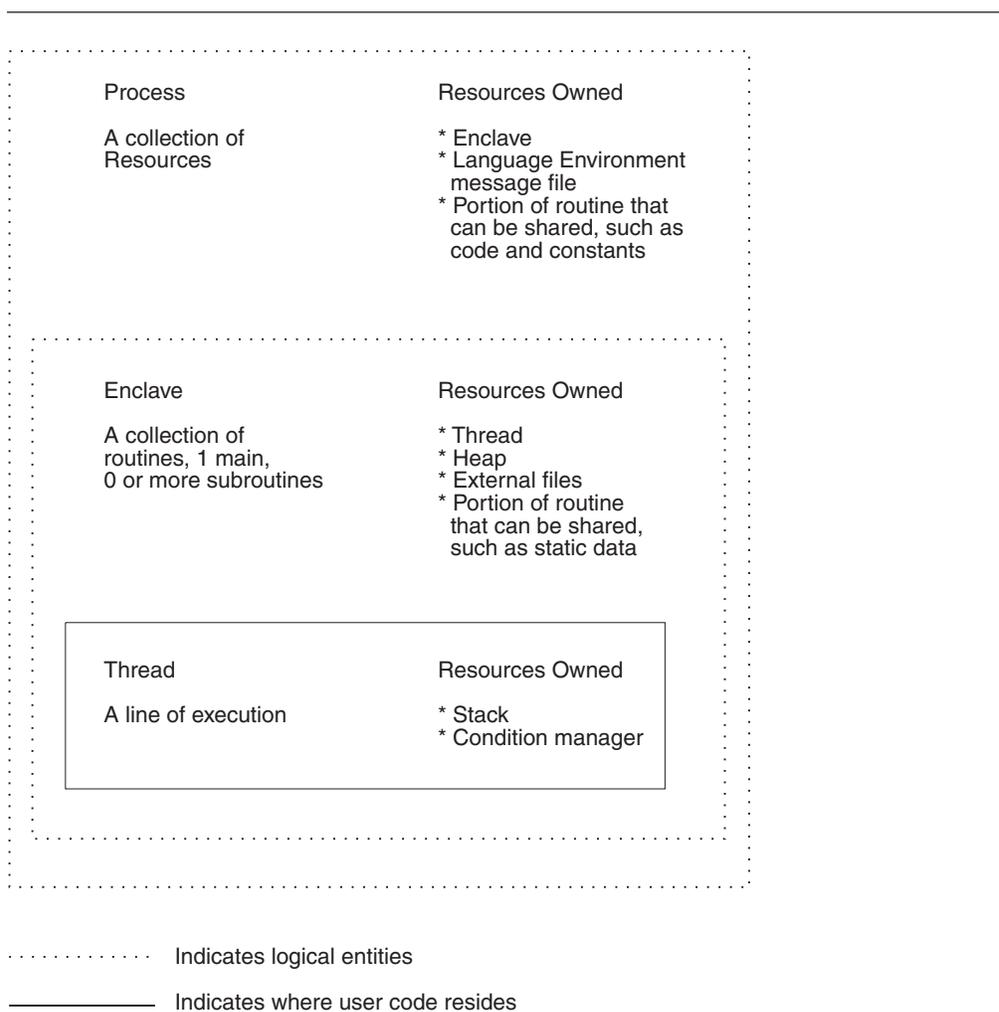


Figure 18. Program model illustration of resource ownership

Process

A *process* is a collection of resources, both application code and data, consisting of one or more related enclaves (described in the next section). The process is the outermost or highest level run-time component of the common run-time environment. The resources maintained at the process level do not affect the language semantics of an application running at the enclave level.

The Language Environment library is an example of the type of resource that is maintained at the process level. The Language Environment library is loaded at process initialization, although it could be loaded for any of the individual enclaves within the process at enclave initialization. The process is used in the same way by all enclaves created within the process. It has no effect on the HLL semantics of applications running within each of the enclaves.

Each process has an address space that is logically separate from those of other processes. Except for communications with each other using certain Language Environment mechanisms, no resources are shared between processes; processes do not share storage, for example. A process can create other processes. However, all processes are independent of one another; they are not hierarchically related.

Although the Language Environment program model supports applications consisting of one or more processes, z/OS Language Environment supports only a single process for each application that runs in the common run-time environment.

Enclave

A key feature of the program model is the *enclave*, which consists of one or more load modules, each containing one or more separately compiled, bound routines. A load module can include HLL routines, assembler routines, and Language Environment routines.

The Enclave Defines the Scope of Language Semantics

By definition, the scope of a language statement is that portion of code in which it has semantic effect. The enclave defines the scope of the language semantics for its component routines. Scope encompasses names, external data sharing, and control statements such as C's `exit()` statement.

The Enclave Defines the Scope of the Definition of the Main Routine and

Subroutines: The enclave boundary defines whether a routine is a *main* routine or a *subroutine*. The first routine to run in the enclave is known as the main routine in Language Environment. All others are designated subroutines of the main routine.

The first routine invoked in the enclave must be capable of being designated main according to the rules of the language of the routine. For example, a main routine in a Language Environment-conforming C or C++ application would be the `main()` routine. All other routines invoked in the enclave must be capable of being a subroutine according to the rules of the languages of the routines.

If a routine is capable of being invoked as either a main or subroutine, and recursive invocations are allowed according to the rules of the language, the routine can be invoked multiple times within the enclave. The first of these invocations could be as a main routine and the others as subroutines.

The Enclave Defines the Scope and Visibility of the Following Types of Data:

- **Automatic data:** Automatic data is allocated with the same value on entry and reentry into a routine if it has been initialized to that value in the semantics of the language used. Values of the data at exit from the routine are not retained for the next entry into the routine. The scope of automatic data is a routine invocation within an enclave.
- **External data:** External data persists over the lifetime of an enclave and retains last-used values whenever a routine is reentered. The scope of external data is that of the enclosing enclave; all routines invoked within the enclave recognize the external data. Examples are C or C++ data objects of extern storage class.

Program model

- **Local data:** The scope of local data is that of the enclosing enclave; however, local data is recognized only by the routine that defines it. Examples are any C variables with block scope.

The Enclave Defines the Scope of Language Statements: The enclave defines the scope of language statements — for example, those that stop execution of the outermost routine within an enclave. C's `exit()` statement is an example of such a statement. When one of these statements is executed, the main routine within the enclave terminates. Thus, the enclave defines the scope of the language statements.

Prior to returning, resources obtained by the routines in the enclave are released and any open files are closed.

Additional Enclave Characteristics

Management of Resources: The enclave manages most Language Environment resources, such as the thread and heap storage. Heap storage, for example, is shared among all threads within an enclave. Allocated heap storage remains allocated until explicitly freed or until the enclave terminates. None of the enclave-managed resources are shared between enclaves.

Multiple Enclaves:

Restriction: z/OS Language Environment provides explicit support for a single enclave within a single process. Language Environment “nested” enclaves are not supported for AMODE 64 applications.

Thread

Within each enclave is a *thread*, the basic run-time path represented by the machine state; conditions raised during execution are isolated to that run-time path.

Threads share all of the resources of an enclave and therefore do not need to selectively create or load new copies of resources, code, or data. Although a thread does not own its storage, it can address all storage within the enclave. All threads are independent of one another and are not related hierarchically. A thread is dispatched with its own run-time stack, instruction counter, registers, and condition handling mechanisms.

Because threads operate with unique run-time stacks, they can run concurrently within an enclave and allocate and free their own storage. Concurrent, or parallel, processing, is useful when code is event-driven, or for improving the performance of a large application.

The full Language Environment program model

Figure 19 on page 77 illustrates the relationship between the various entities that make up the Language Environment program model.

As Figure 19 on page 77 shows, each process exists within its own address space. An enclave consists of one main routine with any number of subroutines. External data is available only within the enclave in which it resides. External data items that happen to be identically named in different enclaves reference distinct storage locations; the scope of external data, as described earlier, is the enclave. Threads running in an AMODE 64 environment cannot create new “nested” enclaves.

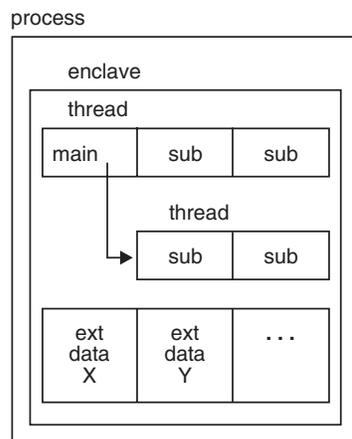


Figure 19. Overview of the Full Language Environment Program Model

Mapping the POSIX program model to the Language Environment program model

Language Environment in conjunction with z/OS UNIX supports POSIX standards (POSIX 1003.1 and POSIX 1003.1c) and the XPG4 standard. The POSIX standard follows a program model which differs somewhat from the Language Environment program model. This section provides a helpful comparison of both models.

The descriptions below are intended to be a brief review for C users of the characteristics of POSIX program entities. For full definitions of these terms, refer to the ISO/IEC9945 for POSIX 1003.1 and POSIX 1003.1c. The XPG4 standard is described in detail in *X/Open Specification Issue 4*.

Key POSIX program entities and Language Environment counterparts

POSIX defines four program model constructs:

Process

An address space, at least one thread of control that executes within that address space, and the thread's or threads' required system resources.

In general, POSIX processes are peers; they run asynchronously and are independent of one other, unless your application logic requests otherwise.

Some aspects of selected processes are hierarchical, however. A C process can create another C process (no ILC is allowed) by calling the `fork()` or `spawn()` functions. Certain function semantics are defined in terms of the parent process (the invoker of the `fork`) and the child process (cloned after the `fork`). For example, when a parent process issues a `wait()` or `waitpid()`, the parent process' logic is influenced by the status of the child process or processes.

A Language Environment process with a single enclave maps approximately to a POSIX process. In Language Environment, starting a main routine creates a new process. In POSIX, issuing a `fork()` or a `spawn()` creates a new process. A POSIX sigaction of `stop`, `terminate`, or `continue` applies to the entire POSIX process.

Program model

A Language Environment process with multiple enclaves is a Language Environment extension to POSIX and is not supported for AMODE 64 applications.

Note: The scope of a specific POSIX function might be the Language Environment process or Language Environment enclave. See “Scope of POSIX semantics” for details.

Process group

Collection of processes. Group membership allows member processes to signal one another, and affects certain termination semantics.

No Language Environment entity maps directly to a POSIX process group.

Session

Collection of process groups. Conceptually, a session corresponds to a logon session at a terminal.

No Language Environment entity maps directly to a POSIX session, but a session is a rough equivalent of a Language Environment application whose execution scope is bound by the end user logon and logoff.

Thread

A single flow of control within a process. Each thread has its own thread ID, state of any timers, *errno* value, thread-specific bindings, and the required system resources to support a flow of control. Threads are independent and not hierarchically related.

A Language Environment thread maps to a POSIX thread. POSIX `pthread_create` creates a new thread under Language Environment.

An enclave that contains multiple threads cannot issue `fork()`, either explicitly or implicitly (`popen()` being mapped to `fork()` and `exec()`).

Scope of POSIX semantics

Some general rules for the scope of POSIX processes follow, as illustrated in Figure 20 on page 79:

- POSIX semantics applied to a POSIX process from outside the POSIX process (interprocess semantics) are applied to a Language Environment process. For example, a signal directed from a process to another process using `kill` is applied to a Language Environment process.
- POSIX semantics scoped to within the current POSIX process (intraprocess semantics) apply to the current Language Environment enclave. For example, heap storage is recognized throughout an enclave.

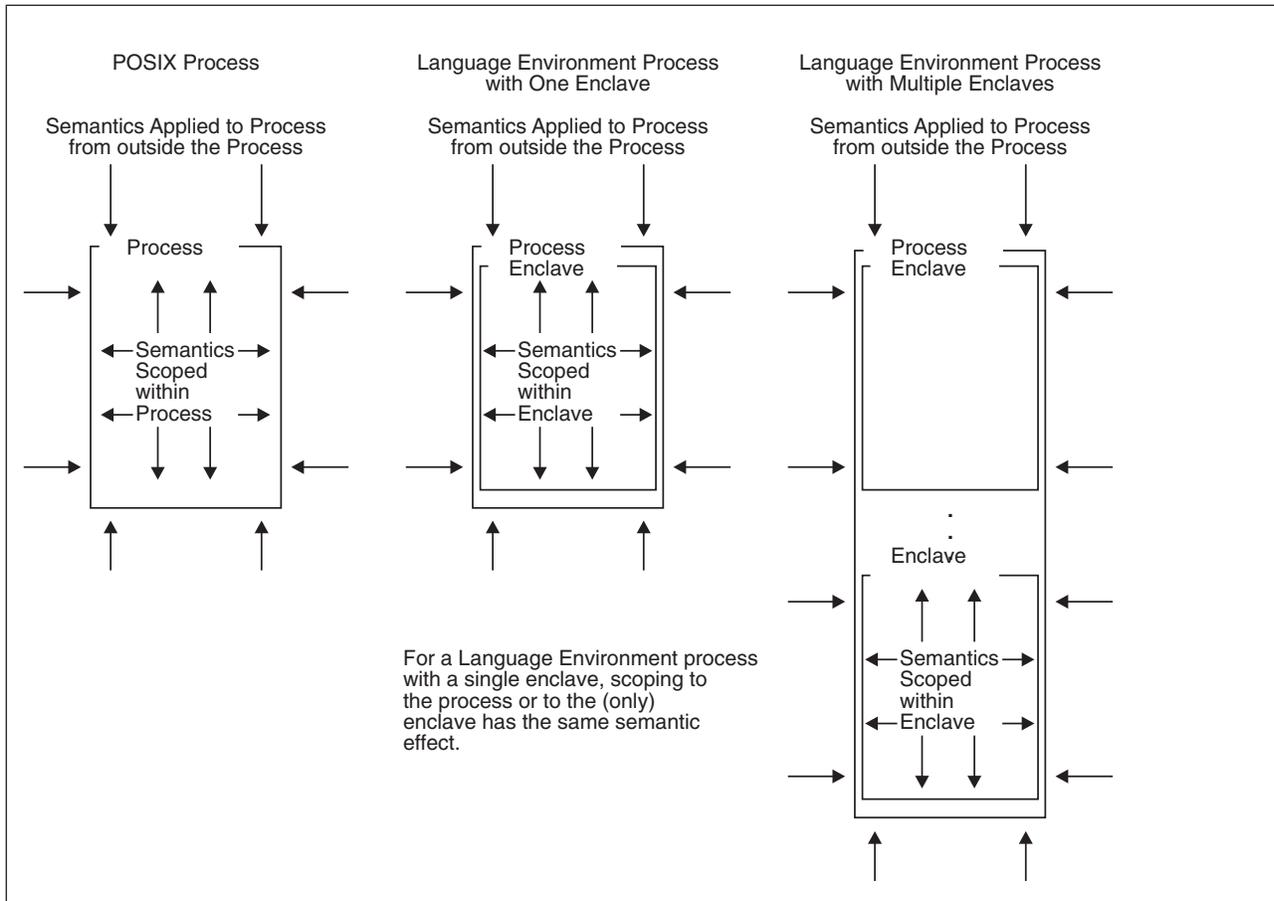


Figure 20. Scope of Semantics against POSIX Processes and Language Environment Processes/Enclaves

Chapter 11. Stack and heap storage

Language Environment provides services that control the stack and heap storage used at run time. Language Environment-conforming HLLs and assembler routines use these services for all storage requests.

Understanding the basics

Language Environment provides the following types of storage:

- Stack storage
Stack storage is automatically created by Language Environment and is used for routine linkage and automatic storage. Refer to “Stack storage overview” on page 82 for more information.
- Heap storage
Heap storage is dynamically allocated at a routine's first request for storage that has a lifetime not related to the execution of the current routine. Refer to “Heap storage overview” on page 84 for more information.

In addition to heap and stack storage, Language Environment provides a function that allows AMODE 64 applications to manipulate memory objects. The `__moservices()` function allows an application to:

- Create a memory object that will be associated with the current Language Environment enclave. The user can request certain attributes to be applied to the memory object, such as the dump priority, and the size of the page frames to be used when backing it.
- Free a memory object that was created using a previous `__moservices()` call
- Specify a shared memory dump priority to be used when allocating shared memory

For more details on `__moservices()`, refer to *z/OS XL C/C++ Run-Time Library Reference*.

Table 13 summarizes the ways in which Language Environment-conforming languages use stack and heap storage. The remainder of this section further discusses stack and heap storage concepts and terminology.

Table 13. Usage of Stack and Heap Storage by Language Environment-Conforming Languages

Language	Stack	Heap
C or C++	<ul style="list-style-type: none">• Automatic variables• Library routines	Variables allocated by: <code>malloc()</code> function <code>__malloc31()</code> function <code>__malloc24()</code> function <code>calloc()</code> function <code>realloc()</code> function Static external (RENT)

Related Options and Services

Runtime Options

HEAP64 Allocates storage for user-controlled dynamically allocated variables

HEAPCHK Specifies that heap storage be inspected for damage

HEAPPOOLS Improves the performance of heap storage allocation above the 16 MB line and below the 2 GB bar

HEAPPOOLS64 Improves the performance of heap storage allocation above the 2 GB bar

IOHEAP64 Allocates I/O-related storage

LIBHEAP64 Allocates library heap storage

RPTSTG Generates a storage report

STACK64 Controls stack allocation above the 2GB bar

STORAGE Controls the initial content of heap and stack

THREADSTACK64 Controls the stack allocation for each thread, except the initial thread, in a multithreaded environment

See *z/OS Language Environment Programming Reference* for syntax information on runtime options.

Stack storage overview

Note: The term *stack*, as used in this chapter, refers to the *user stack*, which is an independent area of stack storage that is located above the 2 GB bar, designed to be used by both library routines and compiled code. All references to *stack storage* and *stack frame* in this chapter are to real storage allocation, as opposed to *invocation stack*, which refers to a conceptual stack.

Stack storage is the storage provided by Language Environment that is needed for routine linkage and any automatic storage. It is a contiguous area of storage obtained directly from the operating system. Stack storage is automatically provided at thread initialization.

A *storage stack* is a data structure that supports procedure or block invocation (call and return). It is used to provide both the storage required for the application initialization and any automatic storage used by the called routine. Each thread has a separate and distinct stack.

The storage stack is divided into smaller segments called *stack frames*, also known as dynamic storage areas (DSAs). A stack frame, or DSA, is dynamically acquired storage composed of a register save area and an area available for dynamic storage allocation for items such as program variables. Stack frames are added to the user stack when a routine is entered, and removed upon exit in a last in, first out (LIFO) manner. Stack frame storage is acquired during the execution of a program

and is allocated every time a procedure, function, or block is entered, as, for example, when a call is made to a Language Environment callable service, and is freed when the procedure or block returns control.

The stack is allocated as one large area of contiguous storage, the size of which is specified by the maximum size parameter of the STACK64 runtime option. Only a portion of the stack is initially available to the application, the amount specified in the initial size parameter of the STACK64 runtime option. The rest of the stack storage is "guarded," which prevents the application from storing into it. When the initial stack area becomes full, a stack increment is created by the operating system by unguarding additional storage contiguous to the currently available stack area. The amount of storage to unguard is specified by the increment size parameter of the STACK64 runtime option.

If the STACK64 option is not specified, Language Environment uses the installation or application defaults. See *z/OS Language Environment Programming Reference* for more information on using the STACK64 runtime option.

Figure 21 shows the standard 64-bit Language Environment stack storage model.

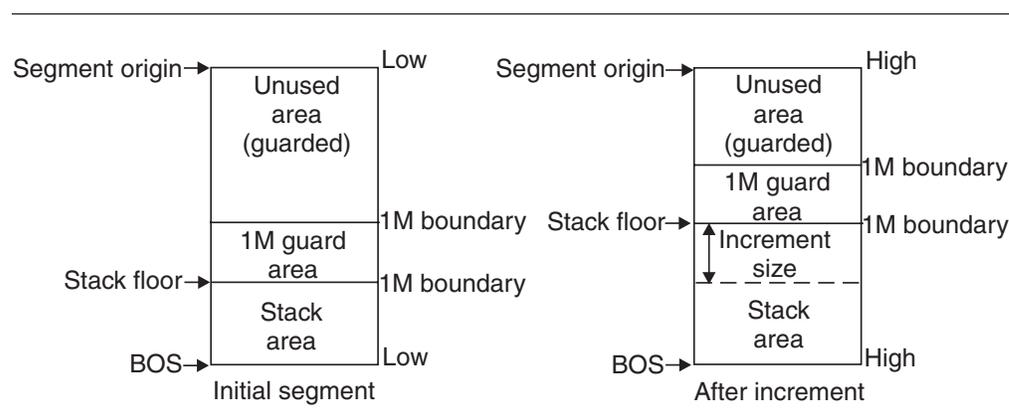


Figure 21. Language Environment Stack Storage Model

Tuning stack storage

For best performance, the initial stack size should be set large enough to satisfy all requests for stack storage. The Language Environment storage report generated by the RPTSTG(ON) option shows you how much stack storage is being used, as well as the total number of stack increments which were required by the application. An initial stack segment that is too large can waste storage and degrade overall system performance.

You can tune stack storage by using the Language Environment STACK64 and THREADSTACK64 runtime options; consult *z/OS Language Environment Programming Reference* for details.

RPTSTG(ON) and the STORAGE runtime option can have a negative affect on the performance of your application, because as the application runs, statistics are kept on storage requests. Therefore, always use the IBM-supplied default setting RPTSTG(OFF) when running production jobs. Use RPTSTG(ON) and STORAGE only when debugging or tuning applications. See *z/OS Language Environment Programming Reference* for more information on RPTSTG and STORAGE.

Stack and Heap Storage

The default value of 128MB for the maximum stack size of the STACK64 and THREADSTACK64 runtime options may cause excessive use of system resources (such as real storage) when running a multithreaded application that creates many pthreads. For such applications, you should use the Language Environment Storage Report (RPTSTG runtime option) to determine your application's actual pthread stack storage usage. Then use the THREADSTACK64 runtime option to set the maximum stack size to a value closer to the actual usage.

Heap storage overview

Heap storage is used to allocate storage that has a lifetime not related to the execution of the current routine. The storage is shared among all program units and all threads in an enclave. (Any thread can free heap storage.) It remains allocated until you explicitly free it or until the enclave terminates.

Heap storage can be allocated or freed several ways. When using C, storage is typically obtained using the `malloc()`, `calloc()`, and `realloc()` functions, and released using the `free()` function; for C++, the `new` and `delete` operators are used. For z/OS Language Environment, heap storage is made up of one or more heap segments comprised of an initial heap segment, and, as needed, one or more heap increments, allocated as additional storage is required. The initial heap may or may not be preallocated prior to the start of the application code, depending on the type of heap. See Figure 22 on page 85 for an illustration of Language Environment heap storage.

Each heap segment is subdivided into individual heap elements. Heap elements are obtained by a call to one of the heap allocation functions, and are allocated within the initial heap segment by the z/OS Language Environment storage management routines. When the initial heap segment becomes full, Language Environment gets another segment, or increment, from the operating system.

There are three basic types of heaps. The “user heap” is the heap storage that is used by the application program, and is obtained and freed by the various C/C++ mechanisms. The “library heap” is the heap storage used internally by Language Environment. In addition, the storage used by Language Environment I/O services is managed separately in the “I/O heap.”

Each of these heaps is subdivided further, based on the requested location of the storage. There is a 64-bit heap for storage allocated above the 2GB bar. For storage allocated below the 16 MB line, there is a 24-bit heap. And, for storage located above the 16 MB line but below the 2GB bar, there is a 31-bit heap. Storage from the below-the-bar locations is useful for communicating with other programs or system services that are not capable of addressing above-the-bar storage. Language Environment provides two heap functions that the application can use to obtain storage from the below-the-bar heaps; `__malloc24()` is used to obtain storage from the 24-bit heap, and `__malloc31()` is used to obtain storage from the 31-bit heap. In both cases, the `free()` function is used to make the storage available again within the heap. For more details on `__malloc24()` and `__malloc31()`, refer to *z/OS XL C/C++ Run-Time Library Reference* .

Language Environment provides runtime options to tune heap storage usage: HEAP64, LIBHEAP64, and IOHEAP64, for the user heap, library heap, and I/O heap, respectively. Within each of these runtime options, you can specify the size of the initial and increment segments for the 64-bit, 31-bit, and 24-bit sections of the heaps. You can also specify the disposition of an increment segment when it is no longer in use; KEEP indicates that the segment remains part of the heap, and

FREE indicates that the segment storage should be returned to the operating system. Note that the initial segment within each heap is never returned to the operating system. See the *z/OS Language Environment Programming Reference* for more details on these runtime options.

You can use the Language Environment STORAGE option to diagnose the use of uninitialized and freed storage.

You can use the HEAPCHK runtime option to run heap storage tests, and to help identify storage leaks.

See Chapter 6, “Using runtime options,” on page 47 and *z/OS Language Environment Programming Reference* for more information on using Language Environment runtime options.

Figure 22 shows the Language Environment heap storage model.

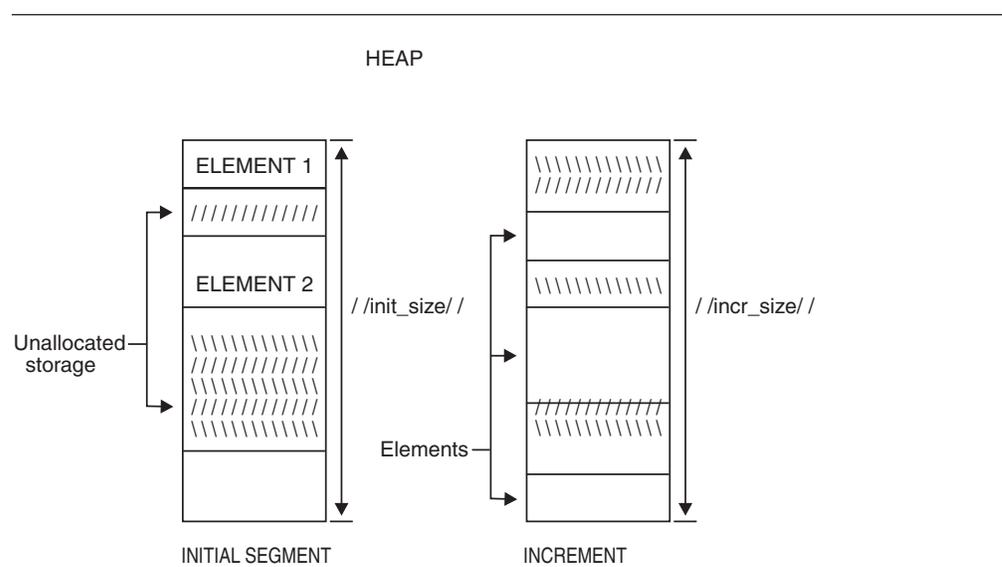


Figure 22. Language Environment Heap Storage Model

Using heap pools to improve performance

Heap pools is an optional storage allocation algorithm for C/C++ applications that is much faster than the normal `malloc()/free()` algorithm in most circumstances. The algorithm is designed to avoid contention for storage in a multithreaded application, and therefore it is important to investigate if your application can benefit from its use.

The heap pools algorithm allows for between one and twelve sizes of storage cells that are allocated from pools out of the heap. For each size, from one to 255 pools can be created where each pool is used by a portion of the threads for allocating storage.

For storage above the 2GB bar, the sizes of the cells, the number of pools for each size, and the sizes of cell pool extents are specified by the `HEAPPOOLS64` runtime option, which is also used to enable the heap pools algorithm. For storage above the 16MB line and below the 2GB bar, the sizes of the cells, the number of pools for each size, and the size of cell pool extents are specified by the `HEAPPOOLS` runtime option, which is also used to enable the heap pools algorithm.

Stack and Heap Storage

Note: Use of the Vendor Heap Manager (VHM) overrides the use of the HEAPPOOLS64 and HEAPPOOLS runtime options.

Applications which should use heap pools

The following types of applications can benefit from the use of heap pools:

- Multi-threaded applications: although single-threaded applications can benefit from the heap pools algorithm, multi-threaded applications can get the most benefit because the proper use of heap pools virtually eliminates contention for heap storage.
- Applications which issue many storage requests with a `malloc()` of 64K bytes or less, because the heap pools algorithm cannot be used for a `malloc()` that is greater than 64K bytes.
- Applications that are not storage constrained: the heap pools algorithm gives up storage for speed. When untuned, the heap pools algorithm uses much more storage than the normal `malloc()/free()` algorithm; when properly tuned it uses only slightly more. Therefore, storage constrained applications should try heap pools, but only if the cell sizes and cell pool counts are carefully tuned. (For tuning information, see “Tuning heap storage” on page 87.) It is possible that some applications running with the heap pools algorithm will have to increase their region size.

Heap pools modes of operation

You can use the HEAPPOOLS64 option in two modes:

ON This mode is selected by specifying the runtime option `HEAPPOOLS64(ON)`. In this mode, cells can be any size between 8 and 64K that is a multiple of 8. This mode avoids contention during storage allocation and release. This mode uses less storage.

ALIGN

This mode is selected by specifying `HEAPPOOLS64(ALIGN)`. In addition to avoiding contention during storage allocation and release, the goal of this mode is to reduce cache contention when two adjacent cells are being updated at the same time. **Only multi-threaded applications will gain additional benefits from using ALIGN mode instead of ON mode.** This mode uses more storage.

You can use the HEAPPOOLS option in two modes:

ON This mode is selected by specifying the runtime option `HEAPPOOLS(ON)`. This mode avoids contention during storage allocation and release. This mode uses less storage.

ALIGN

This mode is selected by specifying `HEAPPOOLS(ALIGN)`. In addition to avoiding contention during storage allocation and release, the goal of this mode is to reduce cache contention when two adjacent cells are being updated at the same time. **Only multi-threaded applications will gain additional benefits from using ALIGN mode instead of ON mode.** This mode uses more storage.

Choosing the number of pools for a cell size: Contention occurs when two or more threads are allocating or freeing cells that are the same size at the same time. Using multiple pools should eliminate some of this contention because only a portions of the threads will be allocating from each pool. For most cell sizes, there is little contention and one pool will be sufficient. However, there may be one or two cell sizes where a lot of successful get heap requests are occurring and the

maximum cells used is high. These sizes may be candidates for multiple pools. Determining the optimum number of pools to use for these cell sizes will involve comparing performance measurements, like throughput, when different values are used for a representative application workload.

Tuning heap storage

For best performance, the initial heap segment should be large enough to satisfy all requests for heap storage. The Language Environment storage report generated by the RPTSTG(ON) runtime option shows you how much heap storage is being used, the total number of segments allocated to the heap, the statistics for the optional heap pools algorithm, and the recommended values for the HEAP64, LIBHEAP64, IOHEAP64, HEAPPOOLS64, and HEAPPOOLS runtime options. You can use this information to tune your application to minimize the number of segments allocated and freed.

The heap pools algorithm (see “Using heap pools to improve performance” on page 85) can be used to significantly increase the performance of heap storage allocation, especially in a multi-threaded application that experiences contention for heap storage. However, if the algorithm is not properly tuned, heap storage could be used inefficiently.

Tuning the HEAPPOOLS64 algorithm for an application is a three step process:

1. Run your application with the runtime options HEAPPOOLS64(ON) or HEAPPOOLS64(ALIGN) as appropriate (using the default cell sizes and counts), and RPTSTG(ON) for some time with a representative application workload. Then examine the "HEAPPOOLS64 Statistics" and "HEAPPOOLS64 Summary" sections of the "Storage Report for Enclave" report.
2. Change the cell sizes in the HEAPPOOLS64 runtime option to the "Suggested Cell Sizes" from the first run. Re-run the application with a representative workload, using the default counts in the HEAPPOOLS64 option. Examine the storage report.
3. The values listed in the "Maximum Cells Used" column of the HEAPPOOLS64 Summary should be the optimal values for the counts to minimize storage use. For a cell size that has multiple pools, the correct value to use is the largest "Maximum Cells Used" value for that size multiplied by the number of pools for that size.

Any time there is a significant change in the workload, repeat these tuning steps to obtain optimal HEAPPOOLS64 values.

Tuning the HEAPPOOLS algorithm for an application is a three-step process:

1. Run your application with the run-time options HEAPPOOLS(ON) or HEAPPOOLS(ALIGN) as appropriate using the following cell sizes and percentages:

```
(8,10,32,10,128,10,256,10,1024,10,2048,10,3072,1,4096,1,
8192,1,16384,1,32768,1,65536,1)
```

and RPTSTG(ON) for some time with a representative application workload. It may be necessary for the application to increase the region size.

2. Change the cell sizes in the HEAPPOOLS runtime option to the "Suggested Cell Sizes" from the first run. Re-run the application with a representative workload, using the default percentages in the HEAPPOOLS option. Examine the storage report.

Stack and Heap Storage

- | 3. The values listed as "Suggested Percentages for Current Cell Sizes" are the
| recommended values to minimize storage usage. These values should be
| evaluated prior to finalizing cell pool sizes.

Any time there is a significant change in the workload, repeat these tuning steps to obtain optimal HEAPPOOLS values.

RPTSTG(ON) and the STORAGE runtime option can have a negative affect on the performance of your application. Therefore, always use the IBM-supplied default setting RPTSTG(OFF) when running production jobs. Use RPTSTG(ON) and STORAGE(xx,xx,xx) only to debug applications. See *z/OS Language Environment Programming Reference* for more information on RPTSTG and STORAGE.

User-created heap storage

Language Environment can also manage, as a heap, storage which is obtained by a C/C++ application. The following functions provide this user-created heap storage capability:

- `_ucreate()` – Creates a heap using storage provided by the caller
- `_umalloc()` – Allocates storage elements from the user-created heap
- `_ufree()` – Returns storage elements to the user-created heap
- `_uheapreport()` – Generates a storage report to assist in tuning the application's use of the user-created heap

This allows the application more flexibility in choosing the attributes of the heap storage. For instance, the storage could be shared memory that is accessed by multiple programs.

For more information about the user-created heap functions, see *z/OS XL C/C++ Run-Time Library Reference*.

Alternative vendor heap manager

Language Environment provides a mechanism such that a vendor can provide an alternative vendor heap manager (VHM) that can be used by Language Environment C/C++ applications. The VHM replaces the `malloc()` (default operator `new` and default operator `new []` are included), `free()` (default operator `delete` and default operator `delete []` are included), `calloc()` and `realloc()` functions.

The VHM will not manage the following:

- user created heaps (`_ucreate`, `_umalloc`, `_ufree`)
- `_malloc24()`, `_malloc31()`
- IOHEAP64
- LIBHEAP64

Using `_CEE_HEAP_MANAGER` to invoke the alternative Vendor Heap Manager

This environment variable is set by the end-user or the application to indicate that the Vendor Heap Manager (VHM), identified by the *dllname*, is to be used to manage the user heap. The format of the environment variable is:

```
_CEE_HEAP_MANAGER=dllname
```

Note: This environment variable must be set using one of the following mechanisms:

- ENVAR runtime option.
- Inside the file specified by the `_CEE_ENVFILE` or `_CEE_ENVFILE_S` environment variable.

Either of these locations is before any user code gets control, meaning prior to the static constructors, and/or main getting control. Setting of this environment variable once the user code has begun execution will not activate the VHM, but the value of the environment variable will be updated.

Stack and Heap Storage

Chapter 12. Language Environment Condition Handling Introduction

This chapter outlines the Language Environment condition handling model in a POSIX(OFF) environment. It describes what constitutes a condition in Language Environment and how Language Environment supplements existing HLL condition handling methods. It also presents several condition handling scenarios to demonstrate how Language Environment condition handling works.

The chapters that follow describe in detail the steps involved in condition handling under Language Environment, HLL-specific condition handling considerations, and Language Environment—POSIX signal handling interactions.

Understanding the Basics

There are two main concepts of Language Environment condition handling: the stack frame-based model and the unique, 16-byte condition token that it provides to communicate information about conditions to Language Environment resources and services.

Language Environment uses stack frames to keep track of a routine's order of execution, and the exception handlers available for each routine. This ensures that conditions can be isolated and handled precisely where they occur in a routine.

One of the most useful features of the condition handling model is the condition token: a 16-byte data type that contains information about each condition. You can use the condition token as a feedback code or to communicate with Language Environment message services. Unlike a return code, which is specific to the caller and callee of a routine, a condition token communicates between all the routines involved in an application. A condition token contains more instance-specific information about a condition than a return code does.

HLL condition handling techniques are discussed in Chapter 13, "Language Environment and HLL Condition Handling Interactions," on page 99.

Related Options and Services

Runtime Options

TRAP Indicates whether Language Environment routines should handle abends and program interrupts.

APIs

__le_cib_get() Returns pointer to the condition information block that is associated with a condition token passed to a user-written condition handler

__set_exception_handler() Registers an Exception Handler function for the current stack frame. Exception Handlers are used to process exceptions at the thread level (unlike signal catchers which process signals at the process level).

__reset_exception_handler() A nonstandard function that unregisters the Exception Handler function, that was previously registered by the `__set_exception_handler()` function, for the current stack frame.

The Stack Frame Model

A stack consists of an ordered set of stack elements, called stack frames, which are managed in a last-in first-out manner. In this book, unqualified references to *stack* mean *invocation stack*. The invocation stack can contain multiple *invocation stack frames*, which represent invocation instances of routines. A stack frame is added to the stack on entry to a routine and removed from the stack on exit from the routine.

The Language Environment condition handling model is based on stack frames, in which condition handling can be different in different stack frames. Another condition handling model is global condition handling, which means that one condition handling mechanism remains in effect for the life of an application.

The following cause a stack frame to be added to the invocation stack:

- A function call in C or C++ that has not been inlined
- Calling a Language Environment or C API
- Invoking C signal catcher
- Invoking Language Environment exception handler
- C++ `throw()`

A stack frame is added to the stack every time a new routine is entered and removed when it is exited. Language Environment uses stack frames to keep track of such things as the routine currently executing, the point at which an error occurs, and the point at which execution should resume after the condition is handled.

Resume Cursor

The resume cursor generally points to the next sequential instruction where a routine would continue running if it were to resume. Initially, the resume cursor is positioned after the machine instruction that caused or signaled the condition. Move the resume cursor using C++ `throw()` and `catch` clauses.

What Is a Condition in Language Environment?

Language Environment defines a *condition* as any event that can require the attention of a running application or the HLL routine supporting the application. A condition is also known as an exception, interrupt, or signal. Language Environment makes it possible to respond to events that in the past might have caused a routine to abend, including hardware-detected errors or operating system-detected errors.

All of the following can generate a condition in Language Environment:

Hardware-detected errors

Also known as program interruptions, these are signaled by the central processing unit. Examples are the fixed-overflow and addressing exceptions. The operating system derives the error codes from the codes defined for the machine on which the application is running. The error codes differ from machine to machine.

Operating system-detected errors

These are software errors and are reported as abends. An example is an OPEN error.

Software-generated signals

Signals are conditions intentionally and explicitly created by Language Environment library routines, or language constructs (such as C's `raise()` or C++ `throw()`).

Under Language Environment, an *exception* is the original event, such as a hardware signal, software-detected event, or user-signaled event, that is a potential condition. Through the enablement step (described briefly in "Steps in Condition Handling" and in detail in Chapter 13, "Language Environment and HLL Condition Handling Interactions," on page 99), Language Environment might deem an exception to be a condition, at which point it can be handled by Language Environment, user-written exception handlers, if they are present, or HLL condition handling semantics.

Steps in Condition Handling

Language Environment condition handling is performed in three distinct steps: the enablement, condition, and termination steps.

Enablement Step

Enablement refers to the determination that an exception should be processed as a condition. The enablement step begins at the time an exception occurs in your application. In general, you are not involved with the enablement step; Language Environment determines which exceptions should be enabled (treated as conditions) and which should be ignored, based on the languages currently active on the stack. If you do not specify explicitly or as a default any of the services or constructs discussed below, the default enablement of your HLL applies.

Condition Handling Introduction

If Language Environment ignores an exception, the exception is not seen as a condition and does not undergo condition handling. Processing resumes at the next sequential instruction.

You can affect the enablement of exceptions in the following ways:

- Set the TRAP runtime option to handle or ignore abends and program checks. See “TRAP Effects on the Condition Handling Process” for more information.
- Disable specific conditions by doing one of the following:
 - Code a construct such as `signal(sigfpe, SIG_IGN)` in a C/C++ function to request that program checks (in this case divide-by-zero) be ignored if they occur in either routine. Execution continues at the next sequential instruction after the one that caused the divide-by-zero condition.

TRAP Effects on the Condition Handling Process

The TRAP runtime option specifies how Language Environment handles abends and program interrupts; TRAP(ON,SPIE) is the IBM-supplied default. For more information about the TRAP runtime option, see *z/OS Language Environment Programming Reference*.

When TRAP(ON,SPIE) is in effect, Language Environment is notified of abends and program interrupts. Language semantics and C/C++ signal handlers can then be invoked to handle semantics, C/C++ signal handlers. An exception to this behavior is that Language Environment cannot handle Sx22 abends, even if TRAP(ON) is specified.

Condition Step

The condition step begins after the enablement step has completed and Language Environment determines that an exception in your application should be handled as a condition. In the simplest form of this step, Language Environment traverses the stack beginning with the stack frame for the routine in which the condition occurred and progresses towards earlier stack frames. Throughout the following discussion, refer to Figure 23.

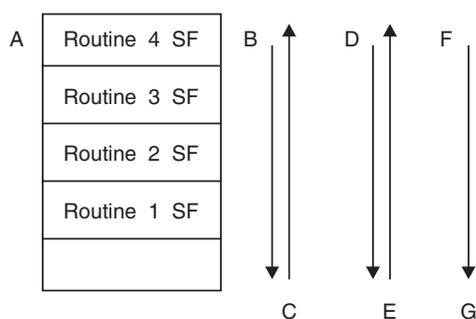


Figure 23. Condition Processing

1. Language Environment condition handling begins at the most recently activated stack frame. This is the stack frame associated with the routine that incurred the condition. In Figure 23, this is **A**, or routine 4.
2. Language Environment traverses the stack, stack frame by stack frame, towards earlier stack frames. This is in the direction of arrow **B** in Figure 23. C/C++

signal handlers or C++ catch clauses can all respond by percolating or handling the condition (see “Responses to Conditions” on page 96 for a discussion of these actions).

3. Condition handling is complete if one of the handlers requests the application to resume execution. If all stack frames have been visited, and no condition handler has requested a resume, the language of the routine in which the exception occurred can enforce default condition handling semantics.
4. Language Environment default actions are then taken based on the severity of the unhandled condition, as indicated in Table 14.

Table 14. Language Environment Default Responses to Unhandled Conditions. Language Environment's default responses to unhandled conditions fall into one of two types, depending on whether the condition was signaled using CEESGL and an fc parameter, or the condition came from any other source.

Severity of Condition	Condition Came from Any Other Source
0 (Informative message)	Resume without issuing message.
1 (Warning Message)	Resume without issuing message.
2 (Program terminated in error)	Terminate the thread. Message issued if TERMTHDACT(MSG) is specified.
3 (Program terminated in severe error)	Terminate the thread. Message issued if TERMTHDACT(MSG) is specified.
4 (Program terminated in critical error)	Terminate the thread. Message issued if TERMTHDACT(MSG) is specified.

Termination Step and the TERMTHDACT Runtime Option

You can use the TERMTHDACT runtime option to set the type of information you receive after your application terminates in response to a severity 2, 3, or 4 condition. For example, you can specify that a message or dump is to be generated if the application terminates.

TERMTHDACT behavior under z/OS UNIX differs slightly; for details, see “Termination Step under z/OS UNIX” on page 109.

Invoking exception handlers

After a condition has been enabled, Language Environment steps through the stack and passes control to the most recently established condition handling routines in the stack. Condition handling routines can be in the form of the user-written exception handlers, or a language-specific condition handling mechanism:

User-Written Exception Handler

See *z/OS XL C/C++ Run-Time Library Reference*.

Language-Specific Condition Handling Semantics

If language-specific semantics are established within a stack frame, they are honored. Of course, the language-specific handling mechanisms act only on those conditions for which the language has a defined action. The language *percolates* all other conditions by passing them on to the next condition handler.

If a condition is unhandled after the stack is traversed, default language-specific and Language Environment condition semantics take over.

Responses to Conditions

exception handlers are routines written to respond to conditions in one of the following ways:

Resume

A resume occurs when a condition handler determines that the condition was handled and normal application execution should resume. A program resumes running usually at the instruction immediately following the point where the condition occurred.

A resume cursor points to the place where a routine should resume.

Percolate

A condition is percolated if a condition handler declines to handle it.

Example: No catch clause at this stack frame.

- With the next condition handler associated with the current stack frame. This can be either the first condition handler in a queue of user-established exception handlers, or the language-specific condition semantics.

Condition Handling Scenarios

The following condition handling scenarios can help you better understand what occurs during the condition handling steps. The scenarios differ in complexity, with Scenario 1 being the easiest to understand.

See Chapter 13, “Language Environment and HLL Condition Handling Interactions,” on page 99 if you are interested in specific HLL condition handling behavior.

Scenario 1: Simple Condition Handling

Refer to Figure 24 throughout the following discussion.

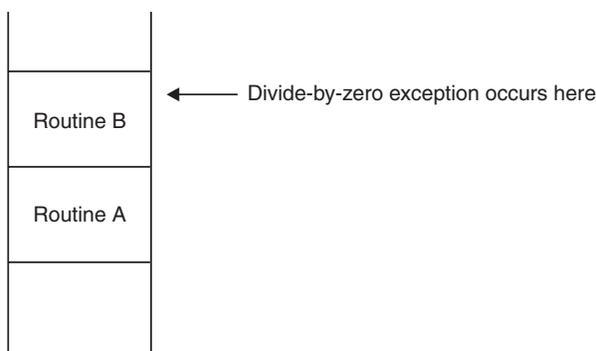


Figure 24. Scenario 1: Division by Zero with No User Exception Handlers Present

In this scenario, there are no C/C++ handlers created by a call to `signal()` or user-written exception handlers.

1. A divide-by-zero exception occurs in routine B.
2. The divide-by-zero exception is enabled by the language of the stack frame in which it occurred because it is a problem that, if it remains unhandled, causes termination.
3. The following occurs in the condition step:

- No handlers have been registered, so the condition is percolated from B's stack frame to A's stack frame.
 - No handlers have been registered, so the condition is percolated.
 - After the oldest stack frame (in this case, that for routine A) has been checked, HLL and Language Environment default actions occur. Assume that the HLL percolates the condition to Language Environment. Language Environment examines the severity of the unhandled divide-by-zero condition (severity 3), so termination step is started.
4. The following occurs during the termination step:
- Language Environment takes the default action for the unhandled condition, which terminates the enclave.

Scenario 2: Exception Handler Present for Divide-by-Zero

Scenario 2 is much the same as scenario 1, except that routine B has a user-written exception handler established to handle the divide-by-zero condition. Refer to Figure 25 throughout the following scenario.

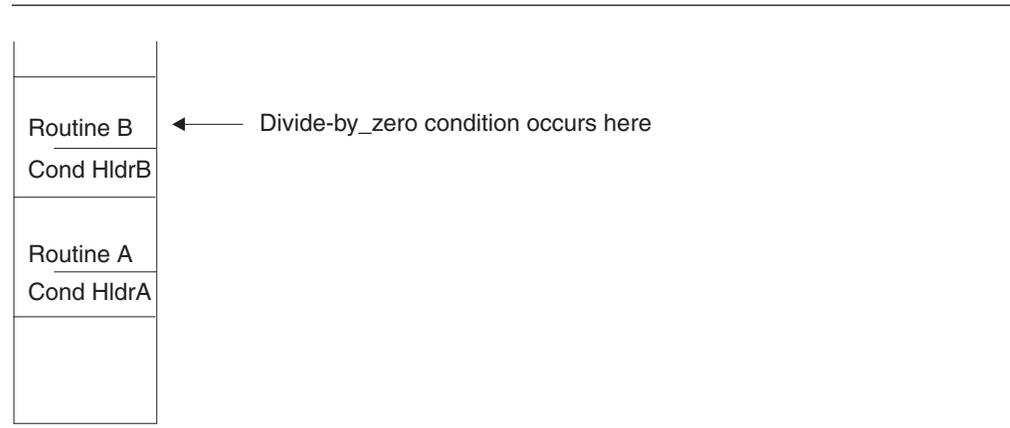


Figure 25. Scenario 2: Division by Zero with a User Handler Present in Routine B

The handler established by routine B is designed to deal with divide-by-zero and possibly other conditions that occur either during its execution or in the routines that it calls. For a divide-by-zero condition, the handler is to print a message and continue processing.

1. A divide-by-zero exception occurs in routine B.
2. The divide-by-zero exception is enabled by the language of the stack frame in which it occurred because it is a problem that, if it remains unhandled, causes termination.
3. The following occurs in the condition step:
 - If a user-written exception handler has been registered using `__set_exception_handler()` on routine B's stack frame, it is given control. The handler recognizes the divide-by-zero as a condition it is capable of dealing with. It produces a message, does appropriate clean-up, and then long jumps back to a recovery point previously established by `setjmp()`.
4. The condition is now considered to be handled and is never seen by Language Environment default handler.

Condition Handling Introduction

Chapter 13. Language Environment and HLL Condition Handling Interactions

This is the second part of the condition handling discussion. It would be helpful for you to read Chapter 12, “Language Environment Condition Handling Introduction,” on page 91 before reading this chapter. Chapter 12 introduces you to terminology and concepts that are discussed in the present chapter, and offers a brief overview of pre-Language Environment HLL condition handling. It discusses in detail the Language Environment condition handling model and the many services that you can use to tailor how conditions are handled in your application. In addition, it introduces the three steps of condition handling in Language Environment.

Understanding the Basics

This chapter discusses HLL condition handling semantics, focusing on how HLL semantics interact with the Language Environment condition handling model and services. C and C++ are each discussed, and condition handling scenarios and examples are provided. This chapter also outlines the interactions between POSIX signal handling and Language Environment condition handling. See one of the following sections for details:

- “C Condition Handling Semantics”
- “C++ Condition Handling Semantics” on page 106
- “Language Environment and POSIX Signal Handling Interactions” on page 106

If you are running a single-language application written in C or C++, which have extensive built-in error handling functions, and you are relying entirely upon the semantics of these languages to handle errors, you will not notice much difference in how errors are handled under Language Environment.

C Condition Handling Semantics

This section describes C condition handling in an POSIX(OFF) environment. If you run applications that contain POSIX functions, you should also read “Language Environment and POSIX Signal Handling Interactions” on page 106, which discusses the interaction between POSIX signal handling and Language Environment condition handling.

C employs a global condition handling model, which, on initialization, defines the actions that are taken when a condition is raised. The actions defined by C apply to an entire enclave, not just to a routine or block within an enclave. You can alter a specific action that the C condition handler takes when a condition is raised, however, by coding `signal()` function calls in your applications.

C recognizes a number of errors; some correspond directly to the errors detected by the hardware or the operating system, and some are unique to C. All actions for condition handling are controlled by the contents of the C global error table. Table 15 on page 100 contains default C-language error handling semantics.

Condition Handling Interactions

Table 15. C Conditions and Default System Actions

C Condition	Origin	Default Action
SIGILL	Execute exception Operation exception Privileged operation raise(SIGILL)	Abnormal termination (return code=3000)
SIGSEGV	Addressing exception Protection exception Specification exception raise(SIGSEGV)	Abnormal termination (return code=3000)
SIGFPE	Data exception Decimal divide Exponent overflow Fixed-point divide Floating-point divide raise(SIGFPE)	Abnormal termination (return code=3000)
SIGABRT	abort() function raise(SIGABRT)	Abnormal termination (return code=2000)
SIGABND	Abend the function	Abnormal termination (return code=3000)
SIGTERM	Termination request raise(SIGTERM)	Abnormal termination (return code = 3000)
SIGINT	Attention condition	Abnormal termination (return code = 3000)
SIGIOERR	I/O errors	Ignore the condition
SIGUSR1	User-defined condition	Abnormal termination (return code=3000)
SIGUSR2	User-defined condition	Abnormal termination (return code=3000)
Masked	Exponent overflow Fixed-point underflow Significance	These exceptions are disabled. They are ignored during the condition handling process, even if you try to enable them using the CEE3SPM callable service.

Comparison of C-Language Environment Terminology

The term *signal* is defined differently under C than under Language Environment, and you need to know the distinction to understand how C and Language Environment condition handling interact. Here is a comparison of the terminology Language Environment and C use to describe the same general idea:

- Using C functions, you *register* a signal handler by using the `signal()` function, and you *raise* a signal using the `raise()` function.

You can think of *signal* as the C term for a Language Environment *condition*. To simplify the following discussion, the term *condition* is used in place of *signal*.

C signal handling functions are recognized in C++ applications. You can write a condition handling routine in C++ using C `signal()` and `raise()` functions. Unique C++ exception handling functions are discussed in “C++ Condition Handling Semantics” on page 106.

Controlling Condition Handling in C

In C, conditions can come from two main sources:

- An exception might occur because of an error in the code. The exception might or might not be seen as a condition, depending on how you use the `signal()` function.
- You can explicitly report a condition by using the `raise()` function.

Using the `signal()` Function

The C `signal()` function call alters the actions that the global error table specifies will be taken for a given condition. You can use `signal()` to do the following:

- Ignore the condition completely. You do this by specifying `signal(sig_num, SIG_IGN)`, where `sig_num` represents the condition to be ignored. When the action for the condition is to ignore it, the condition is considered to be *disabled*. The condition will therefore not be seen.

Note: Exceptions to this rule are the SIGABND condition and the system or user abend represented by Language Environment message number 3250. These are never ignored, even if you specify SIG_IGN in a call to `signal()`.

- Reset condition handling to the defaults shown in Table 15 on page 100. Actions for handling a condition are implicitly reset to the system default when the condition is reported, but at times you need to explicitly reset condition handling. Specify `signal(sig_num, SIG_DFL)`, where `sig_num` is the condition to be reset.
- Call a signal handler to handle the condition. Specify `signal(sig_num, sig_handler)`, where `sig_num` represents the condition to be handled, and `sig_handler` represents a pointer to the user-written function that is called when the condition occurs.

Using the `raise()` Function

When the C `raise()` function is called for any of the conditions listed in Table 15 on page 100, a corresponding Language Environment condition is automatically raised. For detailed descriptions of conditions EDC6000 through EDC6004, see *z/OS XL C/C++ Programming Guide* and *z/OS Language Environment Run-Time Messages*.

C `atexit()` Considerations

In all C applications, the `atexit` list is honored only after all condition handling activity has taken place and all user code is removed from the stack, which invalidates any jump buffer previously established.

With C, you can register a number of routines that gain control during the termination of an enclave. When using the C `atexit()` function, consider the following:

- A C `atexit` routine can nominate only C routines, but those routines can call routines written in other languages.
- User-written exception handlers can be registered while running an `atexit` routine. However, any jump buffers established are invalid.
- If a severity 2 or greater condition arises while running an `atexit` routine and it is unhandled, further `atexit` routines are skipped and the Language Environment environment is terminated.
- A C `exit()` function issued within an `atexit` routine halts all other `atexit` functions.
- If, while running an `atexit` routine, an attempt to register another `atexit` routine is made, the registration is ignored. The `atexit` routine returns a nonzero result indicating a failure to register the routine.

Condition Handling Interactions

C++ supports `atexit()`, but any function pointer input to `atexit()` must be declared as having extern "C" linkage.

C Condition Handling Actions

In this section the condition handling semantics of C-only applications are described as they relate to the Language Environment condition handling model. Condition handling for applications with both C and non-C routines is discussed in *z/OS Language Environment Writing Interlanguage Communication Applications*.

If an exception occurs while a C routine is executing, the following activities are performed:

1. The Language Environment enablement step of condition handling is entered. If the action defined for the exception is to ignore it for one of the following reasons, the condition is disabled. Execution continues at the next sequential instruction after the point where the condition occurred.
 - You have specified `SIG_IGN` in a call to the `signal()` function for any C condition except `SIGABND` or the system or user abend represented by the Language Environment message number 3250.
 - The exception is one of those listed as masked in Table 15 on page 100.
 - You did not specify any action, but the default action for the condition is `SIG_IGN` (see Table 15 on page 100).
2. If `SIG_IGN` is not specified or defaulted for the exception, and the exception is not masked, the Language Environment condition step of condition handling is entered. These activities then occur:
 - If you have registered a signal handler for the condition, that handler is invoked.
 - If the signal handler handles the condition, control returns to the routine in which the condition occurred. If the signal handler cannot handle the condition, it might force termination by issuing `exit()` or `abort()`, or might issue a `longjmp()`.
Condition handling can only continue after a signal handler gains control if you specify `SIG_DFL` in a call to `signal()`.
 - If exception handlers at every stack frame have had a chance to respond to the condition and it still remains unhandled, the Language Environment default actions described in Table 14 on page 95 take place.
3. Language Environment terminates the enclave.

C Condition Handling Examples

The following sections describe various scenarios of condition handling.

Condition Occurs with No Signal Handler Present: The following three figures illustrate how a condition such as a divide-by-zero is handled in a C routine in Language Environment if you do not use any Language Environment callable services, or don't have any user-written exception handlers registered.

There is no signal handler registered for C370C or any of the other C routines, so the condition is percolated through all of the stack frames on the stack. At this point, C default actions take place of percolating the condition to Language Environment. Language Environment takes its default action for an unhandled severity 3 condition and terminates the application.

Figure 26 is a C main routine that calls C370B, a subroutine that passes data to another subroutine, C370C.

```

/*Module/File Name:  EDCMLTA */
/*****
/* Demonstrate a failing C/370 program                               */
/* with multiple active routines                                   */
/* on the stack. The call sequence is as follows:                 */
/* C370A ---> C370B ---> C370C (which does a divide-by-zero)    */
/*****

#include <stdio.h>

int y = 0;
void C370B(void);

int main(void) {

    printf("In Program C370A\n");
    C370B();
}

```

Figure 26. C370A Routine

Figure 27 is a C subroutine that calls C370C, and passes data to it.

```

/*Module/File Name:  EDCMLTB */
/*****
/* This routine is called to pass data forward to C370C.         */
/* C370C will then cause a zero divide.                          */
/*****

#include <stdio.h>

extern int y;
void C370C(int);

void C370B(void) {

    int x;
    printf("In Program C370B\n");
    x = y;
    C370C(x);
}

```

Figure 27. C370B Routine

Figure 28 on page 104 generates a divide-by-zero. The divide-by-zero condition is percolated back to C370B, to C370A, and to Language Environment default behavior.

Condition Handling Interactions

```
/*Module/File Name:  EDCMLTC  */
/*****
/* This routine is called by C370B to generate a zero divide.      */
/*****

#include <stdio.h>

void C370C(int y) {

    printf("In Program C370C\n");
    y = 1/y;
}
```

Figure 28. C370C Routine

Condition Occurs with Signal Handler Present: Figure 29 contains a simple example of a C application in which $y = a/b$ is a mathematical operation. `signal (SIGFPE, c_handler)` is a signal invocation that registers the routine `c_handler()` and gives it control if a floating-point divide exception occurs.

```
/*Module/File Name:  EDCCSIG  */
/*****
/* A routine with a C/370 condition handler registered.          */
/*****

#include <stdio.h>
#include <signal.h>

#ifdef __cplusplus
extern "C" {
#endif
    void c_handler(int);
#ifdef __cplusplus
}
#endif
int main(void) {
    int a=8, b=0, y;
    /* .
        .
        . */
    signal (SIGFPE, c_handler);
    /* .
        .
        . */
    y = a/b;
    /* .
        . */
}

void c_handler(int i)
{
    printf("handled SIGFPE\n");
    /* .
        . */
    return;
}
```

Figure 29. C Condition Handling Example

If $b = 0$, a floating-point divide condition occurs. Language Environment condition handling begins:

- The enablement step occurs.
 - If Table 15 on page 100 indicates that floating-point divide is a masked exception, the exception is ignored. The floating-point divide is not a masked exception, however.
 - If SIG_IGN is specified for the SIGFPE exception in any of the three examples, then the SIGFPE exception is ignored. However, this does not occur.

The floating-point divide condition is enabled and enters the condition step of condition handling.

If none of the above takes place, the condition manager gives the C signal-handler control. This handler in turn invokes `c_handler()` as specified in the `signal()` function in Figure 29 on page 104. Control is then returned to the instruction following the one that caused the condition.

C Signal Representation of S/370 Exceptions

S/370 exceptions and abends are mapped to C signals. Therefore, if both of the following conditions are true, you can apply C signal handling functions to S/370 exceptions and abends:

- You have set the TRAP(ON,SPIE) or the TRAP(ON,NOSPIE) runtime option (Language Environment condition handling is enabled)

C signal representations for the following exceptions are provided in this section:

- For S/370 exceptions generated by the hardware, see Table 16. Some of the exceptions listed in the table can be masked off for normal Language Environment execution.
- For abends, see Table 17 on page 106.

Table 16. Mapping of S/370 Exceptions to C Signals

Interrupt Code	Interrupt Code Description	C Signal Type
01	Operation exception	SIGILL
02	Privileged-operation exception	SIGILL
03	Execution exception	SIGILL
04	Protection exception	SIGSEGV
05	Addressing exception	SIGSEGV
06	Specification exception	SIGILL
07	Data exception	SIGFPE
08	Fixed-point overflow exception	n/a
09	Fixed-point divide exception	SIGFPE
10	Decimal-overflow exception	SIGFPE
11	Decimal-divide exception	SIGFPE
12	Exponent-overflow exception	SIGFPE
13	Exponent-underflow exception	n/a
14	Significance exception	n/a
15	Floating-point divide exception	SIGFPE

Condition Handling Interactions

Table 17 lists the C signal type for abends that can occur under Language Environment.

Table 17. Mapping of Abend Signals to C Signals

Message	Abend Description	C Signal Type
CEE3250	User-initiated abends (SVC 13)	SIGABND
CEE3250	MVS(VSAM or others)-initiated abends	SIGABND
No message delivered	Language Environment abends for severity 4 errors (U40xx)	n/a
No message delivered	Language Environment-initiated abends	n/a

C++ Condition Handling Semantics

C++ includes the C condition handling model and new C++ constructs `throw`, `try`, and `catch`. For more information on these C++ constructs, see *z/OS XL C/C++ Language Reference*. If you use C exception handling constructs (`signal/raise`) in your C++ routine, condition handling will proceed as described in “C Condition Handling Semantics” on page 99. You can use C or C++ condition handling constructs in your C++ applications, but do not mix C constructs with C++ constructs in the same application because undefined behavior could result.

If you use C exception handling, a C++ routine can register a signal handler by coding `signal()` to handle exceptions raised in either a C or a C++ routine. If you use the C++ exception handling model, only C++ routines can catch a thrown object. When a thrown object is handled by a catch clause, execution will continue after the catch clause in the routine. If a thrown object goes unhandled after each stack frame has had a chance to handle it, C++ defines that the `terminate()` function is called. By default, `terminate()` calls `abort()`. You can call the C++ library function `set_terminate()` to register your own function to be called by `terminate`. When `terminate()` finishes calling the user's function, it will call `abort()`.

C routines do not support `try`, `throw`, and `catch`, nor can C routines use `signal()` to register a handler for thrown objects. A C++ routine cannot register a handler via `signal()` to catch thrown objects; it must use catch clauses. `try`, `throw`, and `catch` cannot handle hardware exceptions, nor C, or Language Environment exceptions.

Language Environment and POSIX Signal Handling Interactions

If you want to run an application that uses POSIX signal handling functions under z/OS UNIX, you need to know how Language Environment condition handling might affect your application. For a detailed discussion of POSIX signal handling functions, see *z/OS XL C/C++ Programming Guide*. For details about the Language Environment condition handling model, see Chapter 12, “Language Environment Condition Handling Introduction,” on page 91.

In Language Environment, POSIX signals are distinguished as follows:

Synchronous Signal Handling

If a signal is delivered to the thread that caused the signal to be sent (the *incurring* thread), and the signal is not blocked, Language Environment's synchronous signal handling semantics apply and you can use Language

Environment condition services to handle the condition as described in “Synchronous POSIX Signal and Language Environment Condition Handling Interactions.” Like asynchronous signals, synchronous POSIX signals do not increment the ERRCOUNT error count.

Asynchronous Signal Handling

Asynchronous signals include the following:

- Signals generated because of a `kill()`, `raise()`, `pthread_kill()`, `killpg()` or `sigqueue()` in a multithread environment that are delivered to a thread that did not cause the signal to be sent.
- Signals generated because of a `kill()`, `killpg()` or `sigqueue()` from a different POSIX process.
- All signals that were blocked when first sent, and later unblocked.
- Signals generated by an external interrupt not caused by any specific thread. For example, signals can be generated in response to a command typed in at the terminal.
- SIGCHLD, which is sent to a parent process when one of its child processes terminates.
- Signals, such as SIGALRM, generated by the kernel.

Asynchronous signals are handled according to the semantics defined by POSIX. Language Environment condition handling semantics do not apply; for example, the ERRCOUNT runtime option does not increment its error count when an asynchronous signal is sent.

Synchronous POSIX Signal and Language Environment Condition Handling Interactions

This section discusses how Language Environment processes most synchronous POSIX signals. (In this section, the term *POSIX signal* includes both POSIX-defined signals and C-language signals.) With the exception of the POSIX signals listed in “POSIX Signals that Do Not Enter Condition Handling” on page 109, normal Language Environment condition handling steps occur after a specific thread is selected as the target of a possible signal delivery. This applies whether the signal was directed to a specific thread or to a process (or processes).

Synchronous signal handling takes effect for the following signals, unless they are blocked by the signal mask, or are declared by the arrival of another signal:

- A hardware or software exception caused by a specific thread, which will be delivered to the incurring thread
These are the exceptions typically caught by ESTAE or ESPIE.
- A `kill()` to the current process, a `raise()`, or a `sigqueue()` if the process has but a single thread or the signal happens to be delivered to the thread that issued the `kill()`, `raise()` or `sigqueue()`.
- A `pthread_kill()` issued by a thread to itself

The signal mask is ignored for a signal caused by a program check.

Language Environment processes POSIX signals by using the three general steps of Language Environment condition handling: enablement, condition, and termination, as described below.

Enablement Step for Signals under z/OS UNIX

Figure 30 illustrates how z/OS UNIX determines if a signal is enabled, ignored, or blocked. A few POSIX signals do not go through this process. See “POSIX Signals that Do Not Enter Condition Handling” on page 109 for details.

If a signal is ignored or blocked, the signal does not enter Language Environment synchronous condition handling. If a signal is enabled, z/OS UNIX passes it to the Language Environment enablement step (described in “Enablement Step” on page 93). From there, Language Environment either disables the signal, or passes it into the Language Environment condition step.

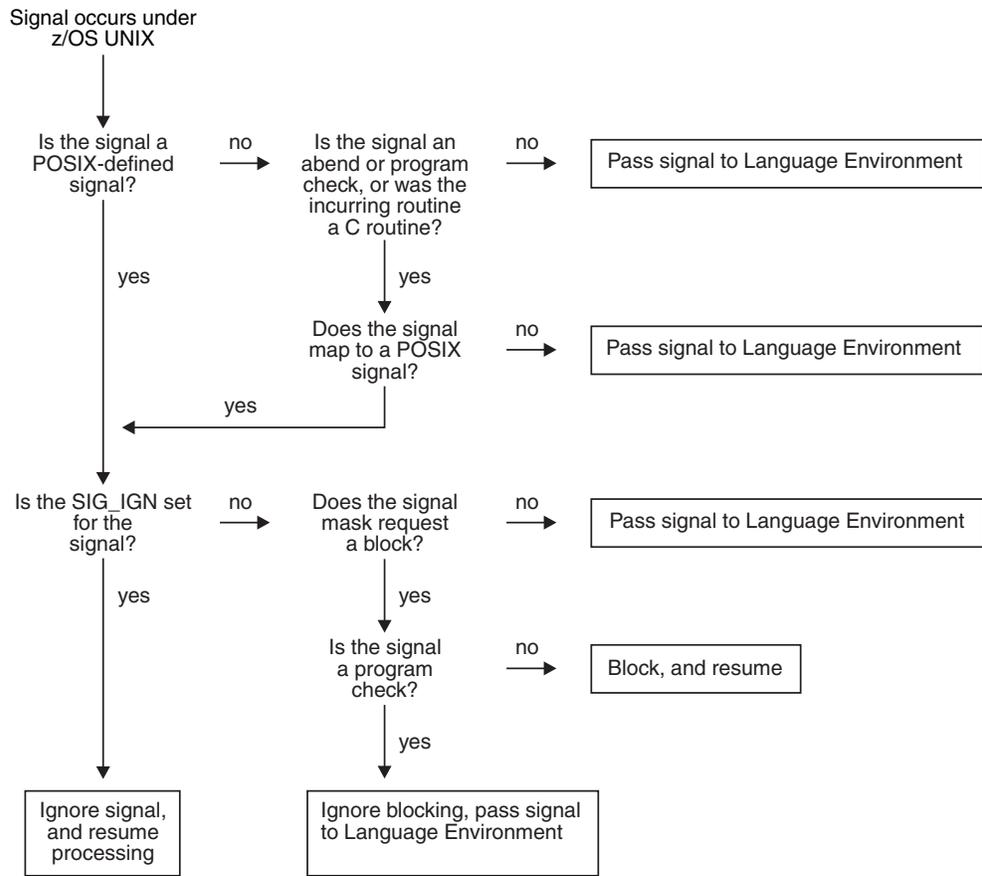


Figure 30. Enablement Step for Signals under z/OS UNIX

Condition Step for POSIX Signals under Language Environment

You might find it helpful to read about the Language Environment condition step before reading this section.

1. At each stack frame (or until the condition is handled, or all of your application's stack frames have been visited), do the following:
 - If the signal action was set in a call to `signal()`, the action requested by the signal handler takes place.
If the signal action was set in a call to `sigaction()`, `sigactionset()` or `bsd_signal()`, the action is ignored until a later step.
2. When all application stack frames have been visited, the incurring stack frame's language defaults are applied.

C applies its default only if the signal action was set in a call to `signal()`. Otherwise, the signal is percolated.

3. If the signal is percolated from the previous step, the following occurs:
 - If the signal is a POSIX signal whose signal action was set in a call to `sigaction()`, `sigactionset()` or `bsd_signal()`, the POSIX action (`SIG_DFL` or a catcher) is applied.
 - For any other signal, Language Environment applies its default actions (described in Table 14 on page 95).

Termination Step under z/OS UNIX

In a POSIX(ON) environment, Language Environment's termination step takes place as described, with one exception: the behavior of the `TERMTHDACT` runtime option. If `POSIX(ON)` is set, `TERMTHDACT` takes effect only if enclave termination results from a program check or abend, not from signal generating functions such as `raise()`, `kill()`, `pthread_kill()`, `killpg()` or `sigqueue()`.

POSIX Signals that Do Not Enter Condition Handling

Certain POSIX signals do not go through the condition handling steps described above:

- `SIGKILL` and `SIGSTOP` cannot be caught or ignored; they always take effect.
- `SIGCONT` immediately begins all stopped threads in a process if `SIG_DFL` is set.
- `SIGTTIN`, `SIGTTOU`, and `SIGSTP` immediately stop all threads in a process if `SIG_DFL` is set.

For IBM extensions to POSIX signals that do not go through condition handling:

- `SIGDUMP` cannot be caught or ignored; it always takes effect.
- `SIGTHSTOP` and `SIGTHCONT` cannot be caught or ignored; they always take effect.
- `SIGTRACE` cannot be caught or ignored; it always takes effect.

Condition Handling Interactions

Chapter 14. Using Condition Tokens

Language Environment uses the 16-byte condition token data type to perform a variety of communication functions. This chapter describes the format of the condition token and its components, and how you can use the condition token to react to conditions and communicate conditions with other routines.

Understanding the Basics

If you provide an *fc* parameter in a call to a Language Environment callable service, the service sets *fc* to a specific value called a condition token and returns it to your application. (See “The Effect of Coding the *fc* Parameter” for more information.)

If you do not specify the *fc* parameter in a call to a Language Environment service, Language Environment generates a condition token for any nonzero condition and passes it to Language Environment condition handling. (See “Effects of Omitting the *fc* Parameter” on page 113 for more information.)

The condition token is used by the routines of your application to communicate with message services, the condition manager, and other routines within the application. For example, you can use it with Language Environment message services to write a diagnostic message associated with a particular condition to a file. You can also determine if a particular condition has occurred by testing the condition token, or a symbolic representation of it. The structure of the condition token is described in “Understanding the Structure of the Condition Token” on page 113, and symbolic feedback codes are discussed in “Using Symbolic Feedback Codes” on page 115.

Language Environment condition tokens contain a 8-byte Instance Specific Information (ISI) token. The ISI token can contain (depending on whether a condition occurred) insert data that further describes the condition and that can be used, for example, to write a specific message to a file. In addition to insert data, the ISI can contain qualifying data (*q_data*) that user-written exception handlers use to identify and react to a specific condition.

Related Services

Language Environment provides callable services to help you construct and decompose your own condition tokens.

`__le_condition_token_build()`

Builds a new condition token in your application.

The Effect of Coding the *fc* Parameter

The feedback code is the last parameter of all Language Environment callable services. C, C++, routines do not have to do so. (See *z/OS Language Environment Programming Reference* for information on how to provide the feedback code parameter in each HLL.) When the *fc* parameter is provided and a condition is raised, the following sequence of events occurs:

Using Condition Tokens

1. The callable service in which the condition occurred builds a condition token for the condition. The condition token is a 16-byte representation of a Language Environment condition. Each condition is associated with a single Language Environment runtime message.
2. The callable service places information into the ISI, which might contain the following:
 - A timestamp
 - Information that is inserted into a message associated with the conditionFor example, you can use the `__le_msg_add_insert()` callable service (see *z/OS Language Environment Programming Reference*) to generate message inserts.
3. If the severity of the detected condition is critical (severity = 4), it is raised directly to the condition manager. Language Environment then processes the condition, as described in “Condition Step” on page 94.
4. If the condition severity is not critical (severity less than 4), the condition token is returned to the routine that called the service.
5. When the condition token is returned to your application, you can use the condition token in the following ways:
 - Ignore it and continue processing.
 - Get, format, and dispatch the message for display using the `__le_msg_get_and_write()` callable service.
 - Store the message in a storage area using the `__le_msg_get()` callable service.
 - Use the `__le_msg_write()` callable service to dispatch a user-defined message string to a destination that you specify.
 - Compare the condition token to one that is known to you so that you can react appropriately. You can test the condition token for success, equivalence or equality.

See *z/OS Language Environment Programming Reference* for more information about Language Environment callable services.

Testing a Condition Token for Success

To test a condition token for success, it is sufficient to determine if the first 4 bytes are zero; if the first 4 bytes are zero, the remainder of the condition token is zero, indicating that a successful call was made to the service.

The Language Environment condition handling model provides two ways you can check for success using the *fc* parameter. You can compare the value returned in *fc* to the symbolic feedback code CEE000, or you can compare it to a 16-byte condition token containing all zeroes coded in your routine. See “Using Symbolic Feedback Codes” on page 115 for details.

You do not necessarily need to check the feedback code after every invocation of a service or to check for success before proceeding with execution. However, if you want to ensure that your application is invoking callable services successfully, test the feedback code after each call to a service.

Testing Condition Tokens for Equivalence

Two condition tokens are equivalent if they represent the same type of condition, even if not necessarily the same instance of the condition. For example, you could have two occurrences of an out-of-storage condition. Though equivalent conditions, they are not necessarily equal because they occur in different locations in your program.

To determine whether two condition tokens are equivalent, compare the first 8 bytes of each condition token to one another. These bytes are static and do not change depending on the given instance of the condition.

You might want to check for equivalence when writing a message about a type of condition that occurs in your application or when registering a condition handling routine to respond to a given type of condition.

There are two ways to check for equivalent condition tokens:

- You can break down the condition token by coding it as a structure and looking at its individual components.
- The easiest way to test for equivalence is to compare the value returned in *fc* with the symbolic feedback code for the condition you are interested in handling. Symbolic feedback codes represent only the first 8 bytes of a 16-byte condition token. See “Using Symbolic Feedback Codes” on page 115 for details.

Testing Condition Tokens for Equality

To determine whether two condition tokens are equal (that is, the same instance or occurrence of the condition token), you must compare all 16 bytes of each condition token with each other. The last 8 bytes can change from instance to instance of a given condition.

The only way to test condition tokens for equality is to compare the value returned in *fc* with another condition token that has either been returned from a call to a service, or that you have coded as a 16-byte condition token in your routine. Symbolic feedback codes are used to test for equivalence; they are not useful in testing for equality because they represent only the first 8 bytes of the condition token.

Effects of Omitting the *fc* Parameter

When a feedback code is not provided, any nonzero condition is raised. Signaled conditions are processed by Language Environment, as described in “Condition Step” on page 94. If the condition remains unhandled at the end of processing, Language Environment takes the Language Environment default action (defined in Table 14 on page 95). The message delivered is the translation of the condition token into English (or another supported national language).

Understanding the Structure of the Condition Token

Figure 31 on page 114 illustrates the structure of the condition token, with bit offsets shown above the components:

Using Condition Tokens

0	-	31	32-33	34 - 36	37 - 39	40 - 63	64 - 127
Condition_ID			Case Number	Severity Number	Control Code	Facility_ID	ISI

For Case 1 condition tokens,
Condition_ID is:

0 - 15 Severity Number	16 - 31 Message Number
------------------------------	------------------------------

For Case 2 condition tokens,
Condition_ID is:

0 - 15 Class Code	16 - 31 Cause Code
-------------------------	--------------------------

A symbolic feedback code represents the first 8 bytes of a condition token. It contains the Condition_ID, Case Number, Severity Number, Control Code, and Facility_ID, whose bit offsets are indicated.

Figure 31. Language Environment Condition Token

Every condition token contains the components indicated in Figure 31:

Condition_ID

A 4-byte identifier that, with the facility ID, describes the condition that the token communicates. The format of Condition_ID depends on whether a Case 1 (service condition) or Case 2 (class/cause code) condition is being represented. Language Environment callable services and most applications can produce Case 1 conditions. Case 2 conditions could be produced by some operating systems and compiler libraries. Language Environment does not produce them directly.

Figure 31 illustrates the format of the Condition_ID for Case 1 and Case 2 conditions.

Case Specifies if the condition token is for a Case 1 or Case 2 condition.

Severity

Specifies the severity of the condition represented by the condition token.

Control

Specifies if the facility ID has been assigned by IBM.

Facility ID

A 3-character alphanumeric string that identifies the product or component of a product that generated the condition; for Language Environment, the facility ID is CEE. Although all Language Environment-conforming HLLs use Language Environment message and condition handling services, the actual runtime messages generated under Language Environment still carry the language identification in the facility ID.

When paired with a message number, a facility ID uniquely identifies a message in the message source file. The facility ID and message number persist throughout an application. This allows the meaning of the condition and its associated message to be determined at any point in the application after a condition has occurred.

If you create a new facility_ID to use with a message source file you created using CEEBLDTX (see "Creating Messages" on page 121), be aware that the facility ID must be part of the message source file name. Therefore, you must follow the naming guidelines to ensure the module name does not abend.

ISI An 8-byte Instance Specific Information token associated with a given instance of the condition. A nonzero ISI token provides instance specific information. The ISI token contains data on message inserts for the message associated with the condition and a `q_data_token` containing 8 bytes of qualifying data. The ISI token is typically built by Language Environment for system or Language Environment-signaled conditions. The `__le_msg_add_insert()` callable service can be used to define the message inserts within the ISI for a condition token.

The message insert information cannot be retrieved directly; however, the entire formatted message with inserts can be formatted and placed in an application-provided character string using `__le_msg_get()`.

Using Symbolic Feedback Codes

Language Environment provides symbolic feedback codes representing the first 8 bytes of a 16-byte condition token. Using Language Environment-provided symbolic feedback codes saves you from having to define an 8-byte condition token in your code whenever you want to check for the occurrence of a condition. Symbolic feedback codes are limited to testing for conditions rather than actual condition instances: no ISI information is tested using symbolic feedback codes because the comparison is only performed against the first 8 bytes of the condition token.

Language Environment provides include files (copy files) that define all Language Environment symbolic feedback codes. See “Including Symbolic Feedback Code Files” for information about Language Environment symbolic feedback code files.

Locating Symbolic Feedback Codes for Conditions

In Language Environment you can locate symbolic feedback codes in the following ways:

- Look in the first column of the symbolic feedback codes table listed after each of the callable services in *z/OS Language Environment Programming Reference*.

Including Symbolic Feedback Code Files

Symbolic feedback codes are provided for Language Environment, C or C++ conditions. The symbolic feedback code files are stored in the SCEESAMP sample library. To use symbolic feedback codes, you must include the symbolic feedback code files in your source code. The symbolic feedback code files have file names of the form `xxxyyyCT`, where:

`xxx`

Indicates the facility ID of the conditions represented in the file. For example, `EDCyyyCT` contains condition tokens for C- or C++-specific conditions (those with the facility ID of EDC).

`xxx` can be CEE (Language Environment) or EDC (C or C++).

`yyy`

Indicates the facility ID of the language in which the declarations are coded.

CT Stands for “condition token.”

To use symbolic feedback codes, include the file in your source code using the appropriate language construct.

Condition Tokens for C Signals under C and C++

You need the condition token representing an event as input to many Language Environment condition and message handling services. C signals have condition token representations that you can use for this purpose. Table 18 contains condition tokens for C signals seen in C or C++ applications not running in a POSIX environment (for example, C or C++ running POSIX(OFF)). The signals listed in Table 18 have a condition token representation with facility ID of EDC.

Table 18. Language Environment Condition Tokens and Non-POSIX C Signals

Severity	Message Number	Symbolic Feedback Code	Case	Severity	Control	ID	Signal Name	Signal Number
3	6000	EDC5RG	1	3	1	EDC	SIGFPE	8
3	6001	EDC5RH	1	3	1	EDC	SIGILL	4
3	6002	EDC5RI	1	3	1	EDC	SIGSEGV	11
3	6003	EDC5RJ	1	3	1	EDC	SIGABND	18
3	6004	EDC5RK	1	3	1	EDC	SIGTERM	15
3	6005	EDC5RL	1	3	1	EDC	SIGINT	2
2	6006	EDC5RM	1	2	1	EDC	SIGABRT	3
3	6007	EDC5RN	1	3	1	EDC	SIGUSR1	16
3	6008	EDC5RO	1	3	1	EDC	SIGUSR2	17
1	6009	EDC5RP	1	1	1	EDC	SIGIOERR	27

Table 19 contains condition token for C signals seen in C applications running POSIX(ON). The signals listed in Table 19 have a condition token representation with facility ID of CEE.

Table 19. Language Environment Condition Tokens and POSIX C Signals

Severity	Message Number	Symbolic Feedback Code	Case	Severity	Control	ID	Signal Name	Signal Number
3	5201	CEE52H	1	3	1	CEE	SIGFPE	8
3	5202	CEE52I	1	3	1	CEE	SIGILL	4
3	5203	CEE52J	1	3	1	CEE	SIGSEGV	11
3	5204	CEE52K	1	3	1	CEE	SIGABND	18
3	5205	CEE52L	1	3	1	CEE	SIGTERM	15
3	5206	CEE52M	1	3	1	CEE	SIGINT	2
2	5207	CEE52N	1	2	1	CEE	SIGABRT	3
3	5208	CEE52O	1	3	1	CEE	SIGUSR1	16
3	5209	CEE52P	1	3	1	CEE	SIGUSR2	17
3	5210	CEE52Q	1	3	1	CEE	SIGHUP	1
3	5211	CEE52R	1	3	1	CEE	SIGSTOP	7
3	5212	CEE52S	1	3	1	CEE	SIGKILL	9
3	5213	CEE52T	1	3	1	CEE	SIGPIPE	13
3	5214	CEE52U	1	3	1	CEE	SIGALRM	14
1	5215	CEE52V	1	1	1	CEE	SIGCONT	19
1	5216	CEE530	1	1	1	CEE	SIGCHLD	20

Table 19. Language Environment Condition Tokens and POSIX C Signals (continued)

Severity	Message Number	Symbolic Feedback Code	Case	Severity	Control	ID	Signal Name	Signal Number
3	5217	CEE531	1	3	1	CEE	SIGTTIN	21
3	5218	CEE532	1	3	1	CEE	SIGTTOU	22
1	5219	CEE533	1	1	1	CEE	SIGIO	23
3	5220	CEE534	1	3	1	CEE	SIGQUIT	24
3	5221	CEE535	1	3	1	CEE	SIGTSTP	25
3	5222	CEE536	1	3	1	CEE	SIGTRAP	26
1	5223	CEE537	1	1	1	CEE	SIGIOERR	27
1	5224	CEE538	1	1	1	CEE	SIGDCE	38
3	5225	CEE539	1	3	1	CEE	SIGPOLL	5
3	5226	CEE53A	1	3	1	CEE	SIGURG	6
3	5227	CEE53B	1	3	1	CEE	SIGBUS	10
3	5228	CEE53C	1	3	1	CEE	SIGSYS	12
1	5229	CEE53D	1	1	1	CEE	SIGWINCH	28
1	5230	CEE53E	1	1	1	CEE	SIGXCPU	29
1	5231	CEE53F	1	1	1	CEE	SIGXFSZ	30
3	5232	CEE53G	1	3	1	CEE	SIGVTALRM	31
3	5233	CEE53H	1	3	1	CEE	SIGPROF	32
	5234	CEE53I	1	1	1	CEE	SIGDUMP	39
	5235	CEE53J	1	1	1	CEE	SIGDANGER	33
	5236	CEE53K	1	1	1	CEE	SIGTHSTOP	34
	5237	CEE53L	1	1	1	CEE	SIGTHCONT	35
	5238	CEE53M	1	1	1	CEE	SIGTRACE	37

q_data Structure for Abends

When Language Environment fields an abend, condition CEE35I (corresponding to message number 3250) is raised. Language Environment provides *q_data* (qualifying data) for system or user abends as part of the ISI token for condition CEE35I. The *q_data* associated with abends is also listed by message number in *z/OS Language Environment Run-Time Messages*.

q_data is comprised of a list of addresses pointing to information that can be used by HLL and user-written exception handlers to react to a condition. The *q_data* structure for an abend is shown in Figure 32 on page 118.

If an abend occurs, Language Environment signals condition CEE35I (corresponding to message number 3250) and builds the *q_data* structure shown in Figure 32 on page 118.

Using Condition Tokens

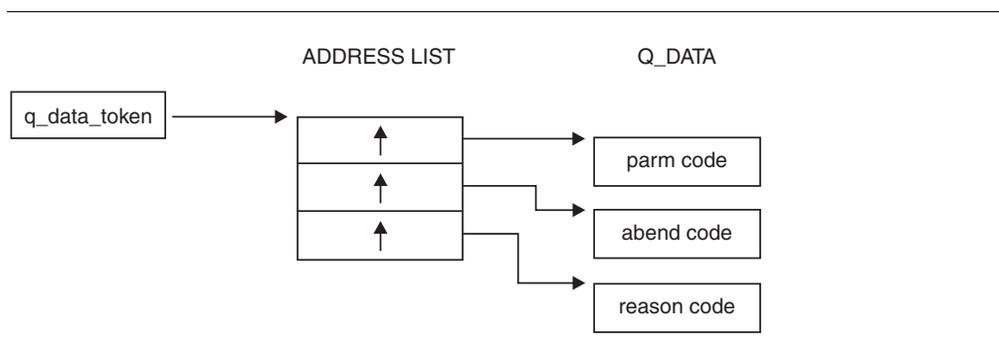


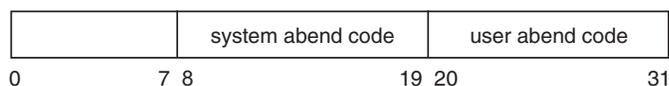
Figure 32. Structure of Abend Qualifying Data

parm count (input)

A fullword field containing the total number of parameters in the `q_data` structure, including `parm count`. In this case, the value of `parm count` is a fullword containing the integer 3.

abend code (input)

A 4-byte field containing the abend code in the following format:



system abend code

The 12-bit system completion (abend) code. If these bits are all zero, then the abend is a user abend.

user abend code

The 12-bit user completion (abend) code. The abend is a user abend when bits 8 through 19 are all zero.

reason code (input)

A 4-byte field containing the reason code accompanying the abend code. If a reason code is not available, `reason code` has the value zero.

q_data Structure for Arithmetic Program Interruptions

If one of the arithmetic program interruptions shown in Table 20 occurs, and the corresponding condition is signaled, Language Environment builds the `q_data` structure shown in Figure 33 on page 119.

Table 20. Arithmetic Program Interruptions and Corresponding Conditions

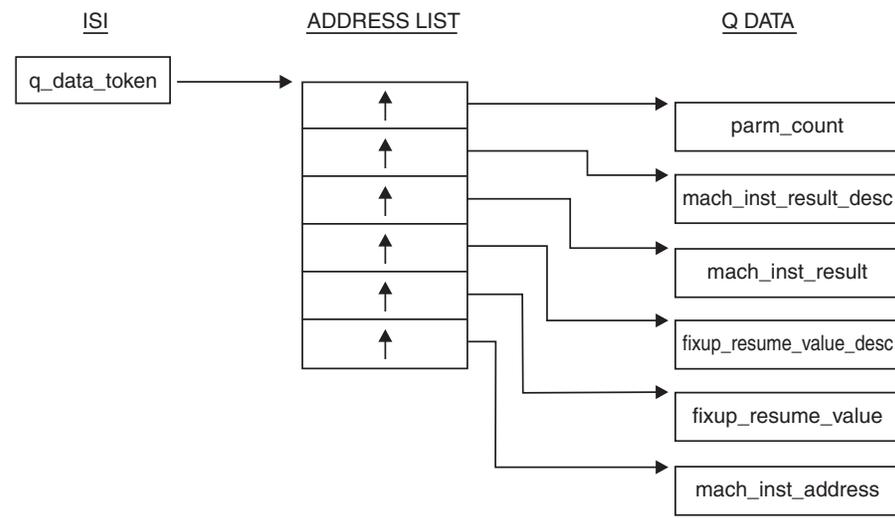
Program Interruption ¹	Program Interruption		Message Number
	Code	Condition	
Fixed-point overflow exception	08	CEE348	3208
Fixed-point divide exception	09	CEE349	3209
Exponent-overflow exception	0C	CEE34C	3212
Exponent-underflow exception	0D	CEE34D	3213
Floating-point divide exception	0F	CEE34F	3215
Unnormalized-operand exception	1E	CEE34U	3230

Table 20. Arithmetic Program Interruptions and Corresponding Conditions (continued)

Program Interruption ¹	Program Interruption Code	Condition	Message Number
-----------------------------------	---------------------------	-----------	----------------

Notes:

1. The square root exception is also an arithmetic program interruption, but is treated like the condition from the square root mathematical routine.

Figure 33. *q_data* Structure for Arithmetic Program Interruption Conditions

The *q_data* structure shown in Figure 33: is built by Language Environment for the conditions of exponent overflow, exponent underflow, floating-point divide, fixed-point overflow, fixed-point divide, and unnormalized-operand exceptions. As a result, the *q_data* structure provides the following information:

parm_count (input)

A 4-byte binary integer containing the value 6, which is the total number of *q_data* fields in the *q_data* structure, including *parm_count*.

mach_inst_result_desc (input)

The *q_data* descriptor for *mach_inst_result*.

mach_inst_result (input)

The value left in the machine register (general register, floating-point register, or element of a vector register) by the failing machine instruction. Based on the program interruption, *mach_inst_result* has one of the following lengths and types (as reflected in the *q_data* descriptor field *mach_inst_result_desc*):

Program Interruption	Length and Type
Fixed-point overflow exception	4- or 8-byte binary integer
Fixed-point divide exception	8-byte binary integer
Exponent-overflow exception	4-, 8-, or 16-byte floating-point number
Exponent-underflow exception	4-, 8-, or 16-byte floating-point number
Floating-point divide exception	4-, 8-, or 16-byte floating-point number
Unnormalized-operand exception (occurs only on vector instructions)	4- or 8-byte floating-point number

Using Condition Tokens

This is also the result value with which execution is resumed when the user condition handler requests the resume action (result code 10).

fixup_resume_value_desc (input)

The `q_data` descriptor for `fixup_resume_value`.

fixup_resume_value (input/output)

The fix-up value which, for the exceptions other than the unnormalized-operand exception, is the result value with which execution is resumed. `fixup_resume_value` initially has one of the following values:

- For an exponent-underflow exception, the value 0
- For an unnormalized-operand exception, the value 0
- For one of the other program interruptions, the same value as in `mach_inst_result`

Based on the program interruption, `fixup_resume_value` has the following lengths and types (as reflected in the `q_data` descriptor field `fixup_resume_value_desc`):

Program Interruption	Length and Type
Fixed-point overflow exception	4- or 8-byte binary integer
Fixed-point divide exception	8-byte binary integer or two 4-byte binary integers (remainder, quotient)
Exponent-overflow exception	4-, 8-, or 16-byte floating-point number
Exponent-underflow exception	4-, 8-, or 16-byte floating-point number
Floating-point divide exception	4-, 8-, or 16-byte floating-point number
Unnormalized-operand exception (occurs only on vector instructions)	4- or 8-byte floating-point number

mach_inst_address (input)

The address of the machine instruction causing the program interruption.

q_data Structure for Square-Root Exception

A *square-root exception* is the program interruption that occurs when a square root instruction is executed with a negative argument. If a square-root exception occurs and the corresponding condition as shown in Table 21 is signaled, Language Environment builds the `q_data` structure shown in Figure 33 on page 119.

Table 21. Square-Root Exception and Corresponding Condition

Program Interruption	Program Interruption Code	Condition	Message Number
Square-root exception	1D	CEE1UQ	2010

For a square-root exception, Language Environment signals the same condition (CEE1UQ) as it does when one of the square root routines detects a negative argument. For this exception, a user-written condition handler can request the same resume and fix-up and resume actions that it can request when the condition is signaled by one of the square root routines.

Chapter 15. Using and Handling Messages

This chapter describes how you can use the Language Environment message services to create, issue, and handle messages for Language Environment-conforming applications.

Understanding the Basics

The Language Environment message services provide a common method of handling and issuing messages for AMODE 64 applications.

When a condition is raised in your application, either Language Environment common routines or language-specific run-time routines can issue messages to stderr. The messages can provide information about the condition and suggest possible solutions to errors.

You can use Language Environment APIs and runtime options to modify message handling.

Related Options and Services

Run-Time Options

NATLANG Specifies the national language run-time message file

APIs

__le_msg_add_insert()

Stores and loads message insert data about a condition

__le_msg_get()

Gets a message

__le_msg_get_and_write()

Gets, formats and writes a message

__le_msg_write()

Writes a message

Utilities

CEEBLDTX Transforms source files into loadable TEXT files

Refer to *z/OS XL C/C++ Run-Time Library Reference* for more information on the APIs.

Creating Messages

The following sections explain how to create messages to use in your routines. To create a message, you:

1. Create a message source file
2. Assemble the message source file with the CEEBLDTX utility
3. Create a message module table
4. Assign values to message inserts
5. Use messages in code to get message output

Creating a Message Source File

The message source file contains the message text and information associated with each message. Standard tags and format are used for message text and different types of message information. The tags and format of the message source files are used by the CEEBLDTX utility to transform the source file into a loadable TEXT file.

Under TSO/E, if you specify a partially qualified name, TSO/E adds the current prefix (usually userid) as the leftmost qualifier and TEXT as the rightmost qualifier. The message source file should have a fixed record format with a record length of 80.

When creating a message file, make sure your sequential numbering attribute is turned off in the editor so that trailing sequence numbers are not generated. Trailing blanks in columns 1–72 are ignored. At least one message data set (TSO/E) is required for each national language version of your messages.

All tags used to create the source file begin with a colon(:), followed by a keyword and a period(.). All tags must begin in column 1, except where noted. Comments in the message source file must begin with a period asterisk (.*) in the leftmost position of the input line.

Figure 34 shows an example of a message source file with a facility ID of XMP.

```
:facid.XMP
:msgno.10
:msgsubid.0001
:msgname.EXMPLMSG
:msgclass.I
:msg.This is an example of an insert,
:tab.+1
:ins 1.a simple insert
:msg., within a message.
:xpl.This is a simple example of how to put an insert into a message.
:presp.No programmer response required.
:sysact.No system action is taken.
```

Figure 34. Example of a Message Source File

The tags used in message source files are:

:facid. The facility ID is required at the beginning of every message file. It is used as the first 3 characters of the message number. All messages within a source file have the same facility ID. For example, all messages issued by Language Environment have a facility ID of CEE. The facility ID is combined with a 4-digit identification number and the message severity code to form the message number. The facility ID can contain any alphanumeric (A–Z, a–z, 0–9) characters.

Omitting the facility ID tag, causes an error during the creation of the loadable message file. Errors are also caused by multiple occurrences of this tag, or by the use of blanks or special characters in the facility ID.

If your C application is running with POSIX(OFF), Language Environment issues messages with a facility ID of EDC for compatibility. For more information, see “Run-Time Messages with POSIX” on page 134.

Note: The facility ID is also used as the first 3 characters of the condition token.

:msgno.

This tag is required. The message number tag defines the beginning and end of information for a message. All information up to the next *:msgno.* tag refers to the current message. The message number appears as the 4 digits following the message prefix, and is used to identify the message in a message source file. Multiple messages can use the same message number, but only if a *:msgsubid.* tag is used within the message.

The message numbers used with the *:msgno.* tags must be in ascending order. The message numbers can be from 1 to 4 numeric (0–9) characters. Leading zeros will be added if fewer than 4 characters are used.

If your application is running with POSIX(ON), message numbers 5201 through 5209 are used whereas the same messages use message numbers 6000 through 6008 when POSIX(OFF) is in effect. For more information, see “Run-Time Messages with POSIX” on page 134.

:msgsubid.

This tag is optional. The message subidentifier tag distinguishes between different messages with the same message number. If every message has a unique message number, the *:msgsubid.* tag is unnecessary.

The numbers associated with the *:msgsubid.* tags must be unique and in ascending order within messages that have the same message number. The number associated with the *:msgsubid.* tag can be from 1 to 4 numeric (0–9) characters. Leading zeros will be added if fewer than 4 digits are used.

:msgname.

The *:msgname.* tag is used to give a name to a message. This name becomes the symbolic name of the condition token associated with the message, and is placed into the COPY file generated by the CEEBLDTX utility. For example, if EXMPLMSG is used for the *:msgname.* tag in a message with a facility ID of XMP, the symbolic feedback code for the condition associated with this message is also EXMPLMSG.

If a message name is omitted, the facility ID plus the base-32 equivalent of the message number is used as the symbolic message name. If additionally the *:msgsubid.* tag is used, the message subidentifier preceded by an underscore is appended to the message name. For example, if *:msgno.* has a value of 10 and the facility ID is XMP, the symbolic feedback code for the condition associated with a message is XMP00A. If additionally the *:msgsubid.* tag is used with a value of 0001, the symbolic feedback code is XMP00A_0001.

:msgclass.

This tag is required. The *:msgclass.* (or *:msgcl.*) tag makes up the final part of the message identification. It requires a case-sensitive character that indicates the severity code of the message. This character corresponds to the level of severity of the condition token associated with the message. If the *:msgclass.* tag differs from the severity level of the condition token, the severity assigned to the condition token is used. Refer to Table 22 on page 133 for the severity codes, levels of severity, and condition descriptions.

:msg.

The *:msg.* tag indicates the beginning of partial or complete text of the message to be displayed. The message text can appear in any national language known to Language Environment (including DBCS characters). For a list of the supported national languages, refer to *z/OS Language*

Using Messages

Environment Programming Reference. The `:msg.` tag can be repeated as often as necessary to construct a message. It is not required if the message consists only of message inserts. If the message text for a message requires more than one line, all lines are left-aligned with the beginning of the first line of message text.

The message text ends with the last nonblank character. There is no fixed space reserved for the message, so there is no requirement to reserve any additional space for message translation.

:hex. The `:hex.` tag indicates the beginning of a hexadecimal character string. If used, it must be within the text of a `:msg.` tag. It is terminated by an `:ehex.` tag. The `:hex.` tag can occur anywhere within the message text.

:ehex. The `:ehex.` tag terminates a string of hexadecimal characters. This tag can occur anywhere within the message text.

:dbc. The `:dbc.` tag defines text of DBCS characters. The string itself cannot contain any SBCS characters, but it must begin with a shift-out character and end with a shift-in character.

:tab.n The `:tab.` tag indicates that the next part of the message will be tabbed over a given number of spaces or tabbed to a given column. If the number is preceded by a plus sign, it indicates the next part of the message will be moved over the specified number of spaces from the current position. Otherwise, the number indicates the column where the next message part will begin. The tab value must be between 1 and 255. If necessary, a new line of output is automatically created to accommodate the tab value. This includes the case where the current position is greater than a specified tab column.

:tbn. The `:tbn.` tag is used to force any text written on a subsequent line to start in the current column until an `:etbn.` tag is found.

:etbn. The `:etbn.` tag turns off the tabs set by a `:tbn.` tag.

:ins n.[text]

The `:ins.` tag defines a message insert. The insert is a variable that is assigned a value with the CEEECMI callable service. The insert number (*n*) can be any number between 1 and 9. The text following the period describes the insert. This text is optional, and is included only in a message file when the value assigned to the insert is not known. For example, the text *variable name* after an insert tag indicates that a variable name is assigned to the insert.

One value can be assigned to each insert used in a message. Insert tags can be moved around, interchanged, or omitted, but the insert values cannot be changed. The order of the `:ins n.` tags, not the insert number, determines the order of the inserts.

:newline.

The `:newline.` tag creates a new message line that can be used for multiline messages.

:xpl. *This tag is optional.* The `:xpl.` tag indicates text used to explain the condition. It is not printed as part of the message, but is included if the message SCRIPT file is formatted and printed.

:presp.

This tag is optional. The `:presp.` tag indicates text that describes the

suggested programmer response. It is not printed as part of the message, but is included if the message SCRIPT file is formatted and printed or displayed online.

:sysact.

This tag is optional. The *:sysact.* tag indicates text that describes the system action. It is not printed as part of the message, but is included if the message SCRIPT file is formatted and printed or displayed online.

Using the CEEBLDTX Utility

z/OS UNIX interface

Syntax

```
ceebldtx  [-C csect_name] [-I secondary_file_name]
          [-P] [-S] [-c class] [-d delimiter]
          [-l BAL | C | COBOL | FORTRAN | PLI] [-s id]
          in_file out_file
```

Note: ceebldtx utility is lowercase in the z/OS UNIX interface and only works with z/OS UNIX files; MVS data sets are not applicable.

Operands

in_file

Required. The name of the file containing the message source.

out_file

Required. The name of the resulting assembler source file containing the messages, inserts, and other items, suitable for input into the High Level Assembler. Extension of ".s" is assumed if none is present.

Options

-C csect_name

This option is used to explicitly specify the CSECT name. An uppercase version of the CSECT name will be used. By default, the CSECT name is the output file base name.

A CSECT name greater than 8 characters requires the use of the GOFF option when assembling the *out_file*.

-I secondary_file_name

The name of the secondary input file generated for the language specified with the *-l* (lowercase L) option. If no suffix is present in the *secondary_file_name* specified, the extension will be ".h" for C, ".fortran" for Fortran, and ".copy" for all others.

-P This option is used to save previous prologs, if files being generated already exist in the directory and contain prologs. By default, previous prologs are not reused.

-S This option is used to indicate sequence numbers should be generated in the files produced. By default, no sequence numbers are generated.

-c class

This option is used to specify the default value for *:msgclass.* in cases where the tag is not coded.

-d APOST | ' | QUOTE | "

This option is used to specify which COBOL delimiter to use. Used in combination with the *-l* (lowercase L) COBOL option. By default, APOST is used as the delimiter.

Using Messages

Note: Quotation marks should be escaped to avoid them being treated as shell meta characters.

Examples:

```
ceebldtx -l COBOL -I secondary_file_name -d \' in_file out_file
ceebldtx -l COBOL -I secondary_file_name -d \" in_file out_file
ceebldtx -l COBOL -I secondary_file_name -d QUOTE in_file out_file
```

-l *BAL* | *C* | *COBOL* | *FORTTRAN* | *PLI*

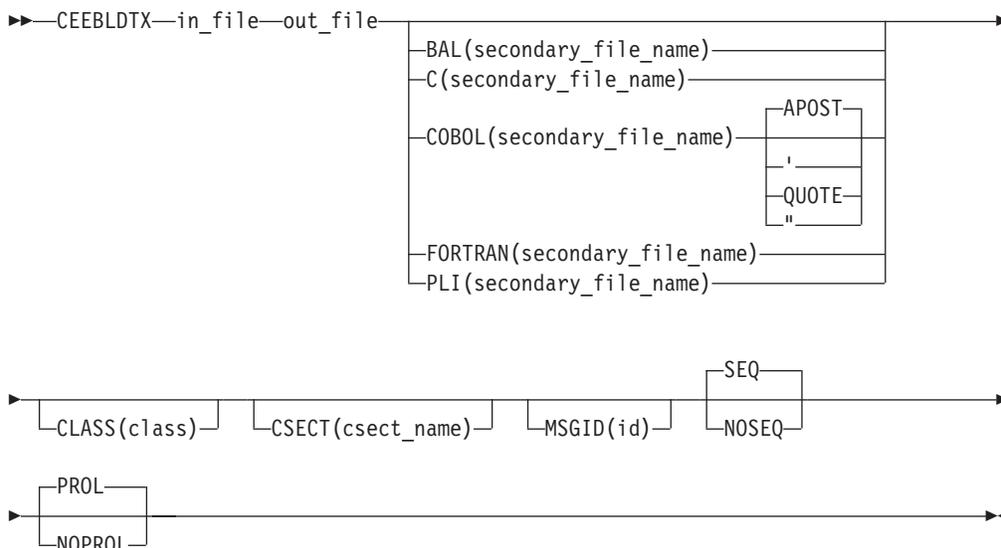
This options is used to specify the language to be used in generating a secondary input file. Used in combination with the -I (uppercase i) secondary_file_name option. The file will contain declarations for the condition tokens associated with each message in the message source file. The language is accepted in lower and upper case.

Note: C370 is also supported.

-s id

This option is used to specify the default value for :msgsubid. in cases where the tag is not coded.

TSO/E interface



in_file

The name of the file containing the message source. The fully qualified data set name must be enclosed in single quotes if you do not want a TSO/E prefix.

out_file

The name of the resulting assembler source file containing the messages, inserts, and other items, suitable for input into the High Level Assembler. The fully qualified data set name must be enclosed in single quotes if you do not want a TSO/E prefix.

options

APOST | *'* | *QUOTE* | *"*

Specify the delimiter to use, APOST is used by default. This option is used to specify which COBOL delimiter to use. Honored in combination with COBOL(secondary_file_name) option.

*BAL(secondary_file_name) | C(secondary_file_name) |
 COBOL(secondary_file_name) | FORTRAN(secondary_file_name) |
 PLI(secondary_file_name)*

secondary_file_name: The name of the secondary input file for the specified language. The file will contain declarations for the condition tokens associated with each message in the message source file. The fully qualified data set name must be enclosed in single quotes if you do not want a TSO/E prefix.

Note:

1. Only the last language (*secondary_file_name*) will be used.
2. C370(*secondary_file_name*) is also supported.

CLASS(class)

This option is used to specify the default value for `:msgclass.` in cases where the tag is not present.

CSECT(csect_name)

This option is used to explicitly specify the CSECT name. An uppercase version of the CSECT name will be used. By default, the CSECT name is the output file base name.

A CSECT name greater than 8 characters requires the use of the GOFF option when assembling the *out_file*.

MSGSID(id)

This option is used to specify the default value for `:msgsubid.` in cases where the tag is not present.

PROL | NOPROL

Specify *PROL* to reuse prolog from the previous file version, if previous version exists. Specify *NOPROL* to ignore the previous prolog. *PROL* is default.

SEQ | NOSEQ

Specify *SEQ* to generate files with sequence numbers. Specify *NOSEQ* to generate files without sequence number. *SEQ* is default.

Note: The CEEBLDTX utility is a REXX EXEC that resides in SCEECLST data set.

Files Created by CEEBLDTX

The CEEBLDTX utility creates several files from the message source file. It creates an assembler source file, which can be assembled into an object ("TEXT") file and link-edited into a module in an MVS load library. When the name of the module is placed in a message module table, the Language Environment message services can dynamically access the messages. See "Creating a Message Module Table" on page 131 for more information about creating a message module table.

The CEEBLDTX utility optionally creates secondary input files (*COPY* or *INCLUDE*), which contain the declarations for the condition tokens associated with each message in the message source file. When a program uses the secondary input file, the condition tokens can then be used to reference the message from the message table. The *:msgname.* tag indicates the symbolic name of the condition token.

To use the CEEBLDTX utility with the sample file shown in Figure 34 on page 122, issue the environment corresponding command example below. After the *out_file* is

Using Messages

generated, High Level Assembler can be used to assemble the out_file into an object and the binder can be used to link-edit the object into a module in an MVS load library. (A CSECT name greater than 8 characters requires the use of the High Level Assembler GOFF option for assembling the primary out_file.):

```
TSO/E:
CEEBLDTX example exmplasm pli(exmplcop)
```

The in_file is EXAMPLE, the out_file is EXMPLASM, and the PL/I secondary input file is EXMPLCOP.

```
z/OS UNIX:
ceebldtx -l PLI -I exmplcop example exmplasm
```

The in_file is example, the out_file is exmplasm.s, and the PL/I secondary input file is exmplcop.copy.

CEEBLDTX Error Messages

Language Environment issues the following messages for CEEBLDTX errors:

Return Code=-1 IRX0005I Machine storage exhausted

Explanation: Rexx terminated execution due to lack of storage. (See IRX0005I in the z/OS TSO/E Messages.)

Programmer response: Attempt one of the following options:

1. Increase the virtual storage space available on the system.
2. Split up the script in_file, into two or more files(Adjust the Message Module Table for the corresponding split.)

Return Code=0005 Error reading file ssssssss.

Explanation: Error occurred while reading file ssssssss.

Programmer response: Validate file accessibility.

Return Code=0006 Error erasing file ssssssss.

Explanation: Error occurred while erasing file ssssssss.

Programmer response: Validate file accessibility.

Return Code=0007 Error writing file ssssssss.

Explanation: Error occurred while writing file ssssssss.

Programmer response: Validate file accessibility.

Return Code=0008 Bad filename ssssssss: forward slash not allowed at the end of a filename.

Explanation: Filenames are not allowed to end with forward slashes.

Programmer response: Modify the filename to not end with a forward slash.

Return Code=0009 Option x requires an argument.

Explanation: Option specified must be accompanied by an argument.

Programmer response: Specify an argument with the option.

Return Code=0010 Invalid option = x. Valid options are: CIPScdls.

Explanation: Invalid option specified.

Programmer response: Specify zero or more valid options.

Return Code=0011 Bad data set name ssssssss.

Explanation: The data set name is not correctly specified.

Programmer response: Validate the name of the data set is correct.

Return Code=0020 CSECT name ssssssss is greater than 63 characters.

Explanation: The CSECT name ssssssss is greater than 63 characters and will cause an error during assembly.

Programmer response: Make sure the CSECT name is 63 characters or less.

Return Code=0021 CSECT name ssssssss does not begin with a letter, \$, #, @ or underscore (_).

Explanation: The CSECT name ssssssss does not begin with a letter, \$, #, @ or underscore (_) and will cause an error during assembly.

Programmer response: Make sure that the CSECT name ssssssss begins with a letter, \$, #, @ or underscore (_).

Return Code=0028 *sssssss* SCRIPT not found on any accessed disk.

Explanation: The SCRIPT file with the name *sssssss* does not exist.

Programmer response: Make sure the name is given correctly and is accessible.

Return Code=0040 Error on line *nnn* in message *nnnn*
Insert number greater than *mmmm*.

Explanation: An insert number greater than the allowable maximum was specified. The current maximum allowable insert number is 9.

Programmer response: Specify an insert number of 9 or less.

Return Code=0044 Error on line *nnn* Duplicate :FACID. tags found with the given script file.

Explanation: Only one facility ID can be specified in the SCRIPT file.

Programmer response: Specify only one facility ID in the SCRIPT file.

Return Code=0048 No :FACID. tag found within the given script file.

Explanation: A 3-character facility ID must be specified in the SCRIPT file with the :facid. tag.

Programmer response: Specify a 3-character facility ID with the :facid. tag.

Return Code=0052 Error on line *nnn* Message number *nnnn* found out of range *mmmm* to *mmmm*.

Explanation: A message was found with a number outside the valid range. The current valid range is 0 to 9999.

Programmer response: Correct the invalid message number on the given line of the SCRIPT file.

Return Code=0056 Number of hex digits not divisible by 2 on line *nnn* in message *nnnn*.

Explanation: Hexadecimal strings must contain an even number of digits.

Programmer response: Specify an even number of digits for the hexadecimal string.

Return Code=0060 Invalid hexadecimal digits on line *nnn* in message *nnnn*.

Explanation: Valid hexadecimal digits are 0–9 and A–F. Invalid digits were detected.

Programmer response: Specify only digits 0–9 and A–F within a hexadecimal string.

Return Code=0064 Number of DBCS bytes not divisible by 2 on line *nnn* in message *nnnn*.

Explanation: Doublebyte character strings must contain an even number of bytes.

Programmer response: Specify an even number of bytes for the doublebyte character string.

Return Code=0068 PLAS out_file name must be longer than the message facility ID *pppp*.

Explanation: The ASSEMBLE file name must be greater than 3 characters.

Programmer response: Specify an ASSEMBLE out_file name of greater than 3 characters.

Return Code=0072 Message facility ID *pppp* on line *nnn* was longer than 4 characters.

Explanation: Facility ID must be exactly 3 characters long, with no blanks.

Programmer response: Specify a 3-character facility ID.

Return Code=0076 Message class on line *nnn* was not a valid message class type: IWESCF A.

Explanation: Message class must be one of the valid message classes.

Programmer response: Specify a valid message class.

Return Code=0080 Tag not recognized on line *nnn*.

Explanation: A tag that was not recognized was encountered.

Programmer response: Check the tag for proper spelling and use.

Return Code=0084 The first tag was not a :FACID. tag on line *nnn*.

Explanation: The first tag of the SCRIPT file must be the facility ID tag.

Programmer response: Specify the facility ID tag as the first tag in the SCRIPT file.

Using Messages

Return Code=0088 Unexpected tag found on line *nnn*.

Explanation: A valid tag was found in an unexpected location in the SCRIPT file; it is likely out of order.

Programmer response: Check the order of the tags in the SCRIPT file.

Return Code=0092 Duplicate tags *ttt* found on line *nnn*.

Explanation: Duplicate :msgname., :msgclass., or :msgsubid. tags were found for a single message.

Programmer response: Remove the extra tag from the message script.

Return Code=0096 No :MSGNO. tags found within the given SCRIPT file.

Explanation: A message file must have at least one message in it, and it must be denoted by a :msgno. tag.

Programmer response: Specify at least one message in the message file.

Return Code=0098 No :MSGCLASS. (or :MSGCL.) tag found for message *nnnn*.

Explanation: A :msgclass. (or :msgcl.) tag was not found for message *nnnn*.

Programmer response: Specify a :msgclass. tag to indicate the severity code of the message and verify the tag is located after the :msgno. tag. Alternatively you can use the -c (CLASS) option to provide a default value for messages which have no :msgclass. (or :msgcl.) tag specified.

Return Code=0100 Insert number was not provided or was less than 1 on line *nnn*.

Explanation: A positive insert number must be provided for each insert.

Programmer response: Specify a positive insert number of 9 or less for the insert.

Return Code=0104 Message subid was out of the range *mmmm* to *mmmm* on line *nnn*.

Explanation: A message subid was found with a number outside the valid range. The current valid range is 0 to 9999.

Programmer response: Correct the invalid message subid on the given line of the SCRIPT file.

Return Code=0108 Existing secondary file, *ssssssss*, found, but not on A-disk.

Explanation: A secondary file with the name *ssssssss* does not exist on A-disk.

Programmer response: Make sure the name is given correctly and is accessible.

Return Code=0112 Current ADDRESS environment not CMS, TSO/E, or z/OS UNIX.

Explanation: CEEBLDTX utility is not being executed in a supported environment.

Programmer response: Transport the utility to either CMS, TSO/E, or z/OS UNIX environment and try executing again.

Return Code=*nnn* Undefined error number *nnn* issued.

Explanation: An undefined error was encountered.

Programmer response: Contact your service representative.

Creating a Message Module Table

Language Environment locates the user-created messages using a message module table that you code in assembler.

The message module table begins with a header that indicates the number of languages in the table. In Figure 35, for example, only English is used, so the first fullword of the header declares the constant F'1'.

```

                TITLE 'UXMPMSGT'
UXMPMSGT CSECT
            DC    F'1'                number of languages
            DC    CL8'ENU            ' language identifier
            DC    A(TABLEENU)        pointer to first language table
TABLEENU DC    F'01'                lowest message number in module
            DC    F'100'            highest message number in module
            DC    CL8'EXMPLASM'      message module name
            DC    F'-1'              flags indicating the last...
            DC    F'-1'              16-byte entry (a dummy entry)...
            DC    CL8'DUMMY'         in the language table
            END    UXMPMSGT
    
```

Figure 35. Example of a Message Module Table with One Language

In the message module table in Figure 36 on page 132, however, English and Japanese are used, so the first fullword of the header declares the constant F'2'. Following the message module table header are tables for each language.

Creating a Message Module Table

```
TITLE 'UZOGMSGT'
UZOGMSGT CSECT
DC F'2'          number of languages
DC CL8'ENU      ' first language identifier
DC A(TABLEENU)  pointer to first language table
DC CL8'JPN      ' second language identifier
DC A(TABLEJPN)  pointer to second language table
TABLEENU DC F'01' lowest message number in first module
DC F'100'       highest message number in first module
DC CL8'ZOGMSGE1' first message module name
DC F'101'       lowest message number in second module
DC F'200'       highest message number in second module
DC CL8'ZOGMSGE2' second message module name

:
DC F'-1'        flags indicating the last...
DC F'-1'        16-byte entry (a dummy entry)...
DC CL8'DUMMY'   in the language table
TABLEJPN DC F'01' lowest message number in first module
DC F'100'       highest message number in first module
DC CL8'ZOGMSGJ1' first message module name
DC F'101'       lowest message number in second module
DC F'200'       highest message number in second module
DC CL8'ZOGMSGJ2' second message module name

:
DC F'-1'        flags indicating the last...
DC F'-1'        16-byte entry (a dummy entry)...
DC CL8'DUMMY'   in the language table
END UZOGMSGT
```

Figure 36. Example of a Message Module Table with Two Languages

Each language table has one or more 16-byte entries that indicate the name of a load module and the range of message numbers the module contains. The first fullword of each 16-byte entry contains the lowest message number within the corresponding module; the second fullword contains the highest message number for that module. The last 8 bytes of each 16-byte entry contain the name of the message module to be loaded. For example, in Figure 36, Japanese messages numbered 101–200 are found in module ZOGMSGJ2. Finally, each language table ends with a dummy 16-byte entry whose first two fullwords contain the flag F'-1' indicating the end of the language table.

Use an 8-character format for the title of the message module table: 'U' (to indicate that the table contains user-created messages), followed by a 3-character facility ID, followed by 'MSGT'. For example, the title of the message module table for messages using a facility ID of XMP would be 'UXMPMSGT' as shown in Figure 35 on page 131; the title of the message module table for messages having a facility ID of ZOG would be 'UZOGMSGT' as shown in Figure 36.

After you create the message module table:

1. Assemble it into a loadable TEXT file using High Level assembler.
2. Store the message module table in a library where it can be dynamically accessed while your routine is running.

Assigning Values to Message Inserts

After you add message insert tags to the message source file, you can use the Language Environment API `__le_msg_add_insert()` to assign values to the inserts.

Values do not need to be assigned to inserts in sequential order. For example, the value of insert 3 can be assigned before the value for insert 1. For more information about `__le_msg_add_insert()` see *z/OS XL C/C++ Run-Time Library Reference*.

Interpreting Run-Time Messages

Runtime messages are designed to provide information about conditions and possible solutions to errors that occur in your routine. Language Environment common routines and language-specific run-time routines issue run-time messages. All run-time messages in Language Environment are comprised of the following:

- A 3-character facility ID used by all messages generated under Language Environment or a particular Language Environment-conforming product. This prefix indicates the Language Environment component that generated the message, and is also the facility ID in the condition token. Language Environment uses the ID of the condition token to write the message associated with the condition. For more information about the condition token, see Chapter 14, “Using Condition Tokens,” on page 111.
- A message number that identifies the message associated with the condition.
- A severity level that indicates the severity of the condition that was raised.

The format of every run-time message is **FFFnnnnx**

FFF

Represents the facility ID. In z/OS, facility IDs that begin with A through I are reserved for IBM use.

nnnn

Represents the message number.

- x Represents the severity code. This character indicates the level of severity (1, 2, 3, or 4) of the message.

Table 22 lists the severity codes, corresponding severity levels, explanations of the severity codes, and the default actions taken if conditions corresponding to each level of severity are unhandled.

Table 22. Language Environment Run-Time Message Severity Codes

Severity Code	Level of Severity	Explanation	Default Action If Condition Unhandled
I	0	An informational message (or, if the entire token is zero, no information).	No message issued.
W	1	A warning message; service completed, probably successfully.	No message issued. Processing continues for all languages.
E	2	Error detected, correction attempted, service completed, perhaps successfully.	Issues message and terminates thread.
S	3	Severe error detected, service incomplete with possible side effects.	Issues message and terminates thread.
C	4	Critical error detected, service incomplete with condition signaled.	Issues message and terminates thread.

Interpreting Run-Time Messages

Language Environment messages can appear even though you made no explicit calls to Language Environment services.

Some Language Environment conditions have qualifying data associated with the instance specific information (ISI) for the condition. For more information about qualifying data, see “q_data Structure for Abends” on page 117.

Specifying National Language

You can use Language Environment national language support to view run-time messages in mixed- and uppercase U.S. English and in Japanese. You can also use national language support to select the most appropriate language variables for your messages, such as language character set, left-to-right text, single-byte character set (SBCS), and double-byte character set (DBCS).

Language Environment message services support requirements for national language support machine-readable information such as message formatting, message delivery, and normalization (removes the adjacent shift-out, shift-in character in order to make DBCS strings as compatible as possible).

The NATLANG runtime option allows you to set the national language used for messages before you run your routine. The default national language is mixed and uppercase U.S. English. See *z/OS Language Environment Programming Reference* for more information on the NATLANG runtime option.

Run-Time Messages with POSIX

When your C application is running with POSIX(ON), some messages have changed both facility ID and message number. Messages that had a facility ID of EDC and ranged from message number 6000 through 6008 prior to running with POSIX(ON) now have a facility ID of CEE and use message numbers 5201 through 5209. Messages 5210 through 5233 are new for POSIX(ON) and thus do not have a corresponding POSIX(OFF) message number, except for message 5223, which has a facility ID of EDC and a message number of 6009 while running with POSIX(OFF). When your C application is running with POSIX(OFF), facility ID EDC is still used for message numbers 6000 through 6009.

If your C application is coded to respond to specific facility IDs or specific message numbers for processing, you must specify POSIX(OFF) to receive the facility ID of EDC and message numbers 6000 through 6009.

Table 23 shows the conditions, their condition numbers, and facility IDs.

Table 23. Condition Tokens with POSIX

Condition Token	Facility ID with POSIX(ON)	Message Number with POSIX(ON)	Facility ID with POSIX(OFF)	Message Number with POSIX(OFF)
SIGFPE	CEE	5201	EDC	6000
SIGILL	CEE	5202	EDC	6001
SIGSEGV	CEE	5203	EDC	6002
SIGABND	CEE	5204	EDC	6003
SIGTERM	CEE	5205	EDC	6004
SIGINT	CEE	5206	EDC	6005
SIGABRT	CEE	5207	EDC	6006
SIGUSR1	CEE	5208	EDC	6007

Table 23. Condition Tokens with POSIX (continued)

Condition Token	Facility ID with POSIX(ON)	Message Number with POSIX(ON)	Facility ID with POSIX(OFF)	Message Number with POSIX(OFF)
SIGUSR2	CEE	5209	EDC	6008
SIGHUP	CEE	5210	na	na
SIGSTOP	CEE	5211	na	na
SIGKILL	CEE	5212	na	na
SIGPIPE	CEE	5213	na	na
SIGALRM	CEE	5214	na	na
SIGCONT	CEE	5215	na	na
SIGCHLD	CEE	5216	na	na
SIGTTIN	CEE	5217	na	na
SIGTTOU	CEE	5218	na	na
SIGIO	CEE	5219	na	na
SIGQUIT	CEE	5220	na	na
SIGTSTP	CEE	5221	na	na
SIGTRAP	CEE	5222	na	na
SIGIOERR	CEE	5223	EDC	6009
SIGDCE	CEE	5224	na	na
SIGPOLL	CEE	5225	na	na
SIGURG	CEE	5226	na	na
SIGBUS	CEE	5227	na	na
SIGSYS	CEE	5228	na	na
SIGWINCH	CEE	5229	na	na
SIGXCPU	CEE	5230	na	na
SIGXFSZ	CEE	5231	na	na
SIGVTALRM	CEE	5232	na	na
SIGPROF	CEE	5233	na	na
SIGDUMP	CEE	5234	na	na
SIGDANGER	CEE	5235	na	na
SIGTHSTOP	CEE	5236	na	na
SIGTHCONT	CEE	5237	na	na
SIGTRACE	CEE	5238	na	na

Handling Message Output

Runtime messages are directed to `stderr`. You may redirect `stderr` using regular C/C++ methods.

Using C or C++ I/O Functions

Runtime messages and `perror()` messages are directed to the `stderr` standard stream output device.

Message output issued by a call to the `printf()` function is directed to `stdout`.

Handling Message Output

You can change the destination of `printf()` output by redirection. For example, `1>&2` on the command line at routine invocation redirects `stdout` to the `stderr` destination.

Table 24 lists the possible destinations of redirected `stderr` and `stdout` standard stream output.

Table 24. C/C++ Redirected Stream Output

	stderr not redirected	stderr redirected to destination other than stdout	stderr redirected to stdout
stdout not redirected	stdout to itself	stdout to itself	Both to stdout
	stderr to itself	stderr to its other destination	
stdout redirected to destination other than stderr	stdout to its other destination	stdout to its other destination	Both to the other stdout destination
	stderr to itself	stderr to its other destination	
stdout redirected to stderr	Both to stderr	Both to the other stderr destination	When stderr and stdout are redirected to each other (this is not recommended), output from both is directed to whichever was specified first.

For more information about redirecting standard streams in C or C++, see *z/OS XL C/C++ Programming Guide*.

Using Multiple Message Handling APIs

See the *z/OS XL C/C++ Run-Time Library Reference* for further details on using the Language Environment message handling APIs.

Chapter 16. Using Date and Time Services

Language Environment for 64-bit Virtual Addressing Mode supports C/C++ and assembler. The C/C++ runtime library provides a large selection of date and time functions. See *z/OS XL C/C++ Run-Time Library Reference* for details.

Chapter 17. National Language Support

Language Environment for 64-bit Virtual Addressing Mode supports C/C++ and assembler. National Language support is provided through runtime option NATLANG and the C/C++ run-time library function `setlocale()`.

Understanding the Basics

National language support services allow you to customize Language Environment output (such as messages, options reports, storage reports, or CEEDUMPs) for a given country by specifying the following:

- The language in which run-time messages are displayed and printed
- The currency symbol, date & time format, and other locale sensitive information

Related Options and Services

Run-Time Options

NATLANG Sets national language for run-time messages

C/C++ APIs

setlocale() Establishes the currency symbol, date & time format, and other locale sensitive information

Setting the National Language

You can set the national language with the NATLANG run-time option. The national language settings affects the error messages. Message translations are provided for the following languages:

ENU Mixed-case U.S. English

UEN Uppercase U.S. English

JPN Japanese

Setting the Locale

See *z/OS XL C/C++ Run-Time Library Reference* and *z/OS XL C/C++ Programming Guide* for more information on `setlocale()` and localization.

Chapter 18. Locale Callable Services

Language Environment for 64-bit Virtual Addressing Mode supports C/C++ and assembler. The C/C++ run-time library provides a large selection of locale-related functions. See *z/OS XL C/C++ Run-Time Library Reference* and *z/OS XL C/C++ Programming Guide* for details.

Chapter 19. General Callable Services

Language Environment for 64-bit Virtual Addressing Mode supports C/C++ and assembler. The C/C++ runtime library provides a set of interfaces that perform general services not directly related to a specific Language Environment function.

Understanding the Basics

The general services are a set of C/C++ APIs that are not directly related to a specific Language Environment function.

Related Services

C/C++ APIs

<code>__cdump()</code>	Generates a dump of the Language Environment runtime environment
<code>__librel()</code>	Returns the C/C++ Runtime Library release level

`__cdump()`

The `__cdump()` function generates a formatted dump of the Language Environment runtime environment. Output from `__cdump()` is normally written to the ddname CEEDUMP. The call to `__cdump()` does not cause your application to terminate. For an example of a formatted dump, see *z/OS Language Environment Debugging Guide*. The `__cdump()` function can be called by your application when you want:

- A trace of calls so you can see the order in which applications were called
- condition information
- entry information
- stack frame contents
- storage around each register (96 bytes)

If your application runs in a non-fork()ed address space, the CEEDUMP DD statement specifies the name of the dump file. If your application runs in the z/OS UNIX environment, the CEEDUMP DD can contain the PATH= keyword, which specifies the fully qualified HFS path and file name.

Specifying a Target Directory for CEEDUMPs

If your application runs in an address space for which you issued a fork() and the CEEDUMP DD data set has not been dynamically allocated, the dump is directed according to the following order:

1. The directory found in environment variable `_CEE_DMPTARG`
2. Your current working directory, if it is not the root directory (/), if this directory is writable, and if the CEEDUMP pathname (made up of the cwd pathname plus the ceedump file name) does not exceed 1024 characters
3. The directory found in environment variable `TMPDIR/` (which specifies the location of a temporary directory other than /tmp)
4. The /tmp directory

General Callable Services

The generated name of the dump is CEEDUMP.*date.time.pid* where *pid* is the z/OS UNIX process ID.

__librel()

The `__librel()` function retrieves the C/C++ Runtime Library release level. The value returned by `__librel()` can be tested to determine if you can use new or extended functions that are available in a particular release. For example, `snprintf()`, `dlopen()`, and `__superkill()` are functions available in release 0x41060000. Before using any of these functions, you can test the Language Environment version to make sure you are running on the release of Language Environment that supports them. See *z/OS XL C/C++ Run-Time Library Reference* for more information on the `__librel()` function.

Chapter 20. Math Services

Language Environment for 64-bit Virtual Addressing Mode supports C/C++ and assembler. The C/C++ run-time library provides a large selection of math functions. See *z/OS XL C/C++ Run-Time Library Reference* and *z/OS XL C/C++ Programming Guide* for details.

Part 4. Specialized Programming Tasks

The chapters in this section describe advanced or specialized tasks that you can perform in Language Environment.

Chapter 21. Assembler Considerations

You can run AMODE 64 applications written in assembler language in Language Environment. AMODE 64 applications written in C or C++ can also call or be called by assembler language applications.

This chapter discusses considerations for assembler applications.

You can write assembler language applications that conform to the 64-bit XPLINK call linkage. *z/OS Language Environment Vendor Interfaces* has details on the 64-bit XPLINK architecture that will be useful to an assembler programmer.

Understanding the Basics

Whether you plan to execute a single-language assembler application or a multiple-language application containing assembler code, there are a number of restrictions you must follow under Language Environment.

For example, to communicate with Language Environment and other applications running in the common run-time environment, your assembler application must preserve the use of certain registers and storage areas in a consistent way. Calling conventions for AMODE 64 assembler programs must follow the 64-bit XPLINK linkage conventions. In addition, your assembler program is restricted from using some operating system services. These conventions and restrictions are described in this chapter.

Compatibility Considerations

If you are coding a new assembler routine that you want to conform to the Language Environment interface for AMODE 64 applications or if your assembler routine calls Language Environment services, you must use the macros provided by Language Environment. For a list of these macros, see “Assembler Macros” on page 151. Throughout this book, *Language Environment-conforming assembler routine* refers to an assembler routine coded using the CELQPRLG and associated macros.

Save Areas

Any AMODE 64 assembler routine used within the scope of a Language Environment application must use 64-bit XPLINK save area conventions.

Notes:

1. Language Environment-conforming AMODE 64 assembler main routines are not supported.
2. Language Environment does not support the linkage stack.

Register Conventions

To communicate properly with assembler routines, you must observe certain register conventions on entry into the assembler routine (while it runs), and on exit from the assembler routine.

Language Environment-conforming assembler

When you use the macros listed in “Assembler Macros” on page 151 to write your Language Environment-conforming assembler routines, the macros generate code that follows the required register conventions.

Assembler Considerations

On entry into the Language Environment-conforming AMODE 64 assembler main routine, registers must contain the following values because they are passed without change to the CELQPRLG macro:

R0	Undefined
R1-R3	General registers 1 through 3 may be used to pass part of the parameter list. If they are not used to pass parameters, then they are undefined.
R4	Caller's 64-bit XPLINK register save area (biased by 2K)
R5	Address of called routine's 64-bit XPLINK environment
R6	Undefined
R7	Return address
All others	Undefined

On entry into a Language Environment-conforming AMODE 64 assembler routine, CELQPRLG stores the caller's registers (R4 through R15) in its own DSA.

At all times while the Language Environment-conforming AMODE 64 assembler routine is running, R4 must point to 2K (2048) bytes before the routine's DSA. This is the concept of a "biased" stack pointer.

On exit from a Language Environment-conforming AMODE 64 assembler routine, these registers contain:

R0	Undefined
R1-R3	General registers 1 through 3 may be used to return part of the return value to the calling program. If they do not contain return value information, then they are undefined.
All others	The contents they had upon entry

Considerations for Coding or Running assembler Routines

This section summarizes some areas you might need to consider when coding or running an assembler routine under Language Environment.

GOFF

Language Environment-conforming AMODE 64 assembler routines must be assembled using the GOFF option. The Language Environment macros described in this chapter will emit an error if GOFF is not specified.

Asynchronous Interrupts

If an asynchronous signal is being delivered to a thread running with POSIX(ON), the thread is interrupted for the signal only when the execution is:

- In a user C or C++ routine
- Just prior to a return to a C or C++ routine
- Just prior to an invocation of a Language Environment library from a user routine

C or C++ routines may need to protect against asynchronous signals based on the application logic including the possible use of the POSIX signal-blocking function that is available.

Condition Handling

Language Environment default condition handling actions occur for assembler routines unless you have registered a user-written condition handler using

`__set_exception_handler()` (see *z/OS XL C/C++ Run-Time Library Reference* for more information about `__set_exception_handler()`).

Language Environment relinquishes all enclave-level resources that were obtained by Language Environment when the enclave terminates, and all process-level resources when the process terminates.

Access to the Inbound Parameter String

You can access the 64-bit XPLINK form of the inbound parameter list for the AMODE 64 assembler routine in the calling routine's save area any time after routine initialization. CELQPRLG will always save general registers 1 through 3 in the first 3 words of the "argument area" in the caller's save area so that the parameter list is complete.

Requirement: All parameters passed to and from AMODE 64 assembler code be passed by reference.

CELQSTRT, CELQMAIN, CELQFMAN

Assembler programs cannot call or use directly CELQSTRT, CELQMAIN, or CELQFMAN as a standard entry point. Results are unpredictable if this rule is violated.

When binding an AMODE 64 application it must be possible for the binder to resolve CELQSTRT. As long as the NCAL bindor option is not specified, CELQSTRT will be automatically resolved. If NCAL is used, it becomes necessary to explicitly include CELQSTRT in the bind process.

Mode Considerations

The CELQPRLG macro automatically sets the module to AMODE 64 and RMODE ANY. There is no support for mixing AMODE 31 or AMODE 24 routines in the same Language Environment process.

Language Environment Library Routine Retention (LRR)

Language Environment library routine retention is not supported for AMODE 64 applications.

Assembler Macros

Language Environment provides the following macros to assist in the entry, calling, and exit of AMODE 64 assembler routines, to map external control blocks, to create assembler DLLs, and to use DLLs from assembler routines:

These macro definitions can be found in the Language Environment SCEEMAC data set, which should be specified on the SYSLIB DD statement when assembling your AMODE 64 program.

- CELQPRLG generates a Language Environment-conforming AMODE 64 prolog. You must use CELQPRLG in conjunction with the following macros (see page 152 for syntax).
- CELQEPLG generates a Language Environment-conforming AMODE 64 epilog and terminates the assembler routine (see page 154 for syntax).
- CEERCB generates a RCB mapping (see page 154 for syntax).
- CEEPCB generates a PCB mapping (see page 154 for syntax).

Assembler Considerations

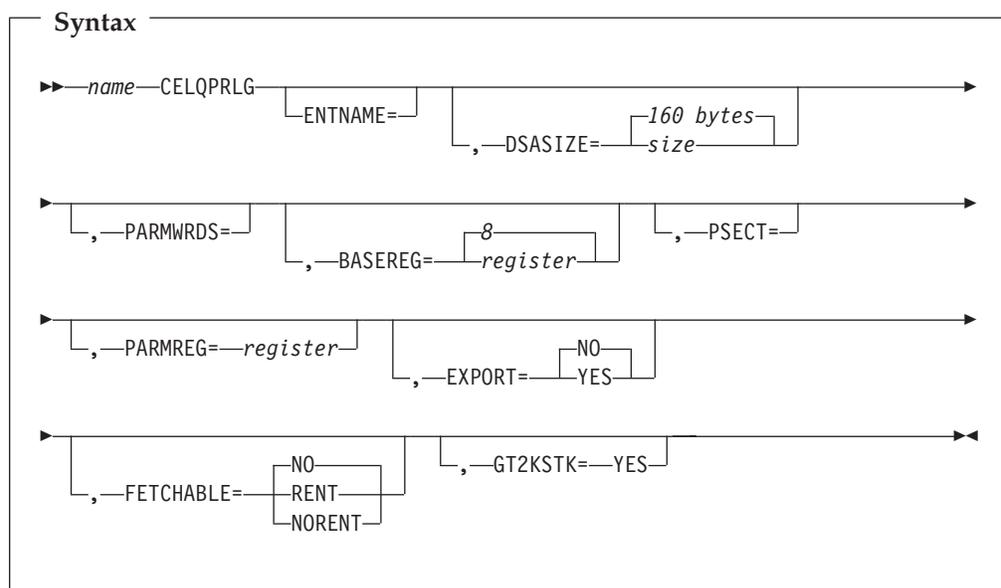
- CEEEDB generates a EDB mapping (see page 154 for syntax).
- CEELAA generates a LAA mapping (see page 155 for syntax).
- CEELCA generates a LCA mapping (see page 155 for syntax).
- CEECAA generates a CAA mapping (see page 155 for syntax).
- CEEDSA generates a DSA mapping (see page 155 for syntax).
- CEEDIA generates a DIA mapping (see page 156 for syntax).
- CELQCALL calls a Language Environment-conforming AMODE 64 routine. It is similar to the CALL macro, except that it supports dynamic calls to AMODE 64 routines in a DLL (see page 156 for syntax).
- CEEPDDA defines a data item in WSA, or declares a reference to an imported data item (see page 158 for syntax).
- CEEPLDA returns the address of a data item defined by CEEPDDA. It is intended to be used to get the address of imported or exported variables residing in the Writeable Static Area (WSA) (see page 159 for syntax).

CELQPRLG Macro — Generate a Language Environment-Conforming AMODE 64 Prolog

CELQPRLG provides a Language Environment-conforming prolog for AMODE 64 routines. The macro generates reentrant code.

You must use CELQPRLG in conjunction with the CELQEPLG macro.

CELQPRLG assumes that the registers contain what is described in “Register Conventions” on page 149 for AMODE 64 assembler routines.



name If ENTNAME=epname is specified, then *name* is used as the name of the 64-bit XPLINK entry marker, else *name* is the name of the entry point and *name#C* is used as the name of the 64-bit XPLINK entry marker.

ENTNAME The optional name of the entry point.

DSASIZE The amount of space used by prolog code for the 64-bit XPLINK

DSA (excluding the 2K bias), the largest parameter list built by this routine, and local automatic variables that are to be allocated for the duration of this routine. This value will be rounded up to a multiple of 32-bytes. If unspecified, the size of the automatic area is the size of a fixed portion of the 64-bit XPLINK DSA without any argument area or automatic variables (160 bytes). This is indicated by the label CEEDSAHPSZ (in the DSA mapping generated by the CEEDSA macro. See “CEEDSA Macro — Generate a DSA Mapping” on page 155 for syntax), and is also the minimum required size.

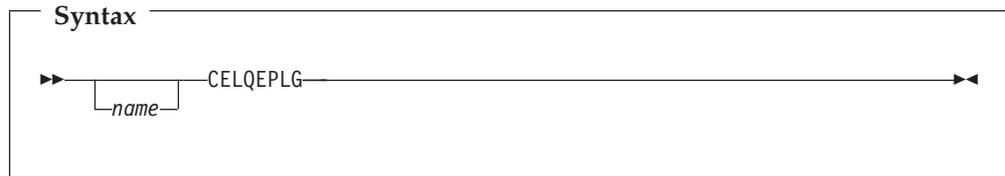
	PARMWRDS	Specifies the number of 4-byte words in the input parameter list. If this is omitted, then the routine will be treated as <i>vararg</i> .
	BASEREG	Designates the required base register. The macro generates code needed for setting the value of the register and for establishing addressability. The default is register 8. If register equals NONE, no code is generated for establishing addressability.
	PSECT	The name to be assigned to the AMODE 64 assembler routine PSECT area. The PSECT is used to establish this routines 64-bit XPLINK environment. For more information about the PSECT area, see <i>HLASM Language Reference</i> .
	PARMREG	Specifies the register to hold the address of the argument area in the caller's save area.
	EXPORT	Indicates whether this entry point will be exported. For EXPORT=NO this entry point can only be called from other routines that are bound into the same program object. For EXPORT=YES, this entry point will be marked as an exported DLL function. If you want the exported name to be a long name and/or mixed case, follow the CELQPRLG macro with an ALIAS statement. For more details on DLLs, including full sample assembler DLL routines, see 13.
		For the extry point to be available as an exported DLL function, you must specify the DYNAM(DLL) binder option, and the resulting program object must reside in a PDSE.
	FETCHABLE	If NO is specified, then this entry point will not be marked as fetchable. If RENT or NORENT is specified, then the CELQEPLG macro will generate either a reentrant or non-reentrant CELQFMAN structure, respectively, so that this entry point can be fetched.
	GT2KSTK	If YES is specified, then an unconditional “large stack frame” prolog will be used that checks for the 64-bit XPLINK stack floor in the LAA, instead of depending on the write-protected guard page. This parameter must be specified if DSASIZE is greater than 2048 (i.e. the 2K stack bias).

Usage Notes:

1. The CELQPRLG macro automatically sets the module to AMODE 64 and RMODE ANY.
2. Unless otherwise indicated, no register values should be expected to remain unchanged after the code generated by CELQPRLG has executed.
3. When more than one CELQPRLG macro invocation occurs in an assembly, it is the programmer's responsibility to code DROP statements for the base registers set up by the previous invocation of the CELQPRLG macro.

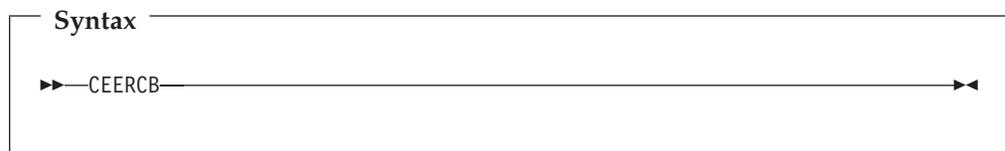
CELQEPLG Macro — Terminate a Language Environment-Conforming AMODE 64 Routine

CELQEPLG provides a Language Environment-conforming epilog and is used to terminate, or return from, a Language Environment-conforming routine. This macro will also generate the necessary structures (e.g. PPA1, PPA2, CELQFMAN, etc.) for the AMODE 64 application.



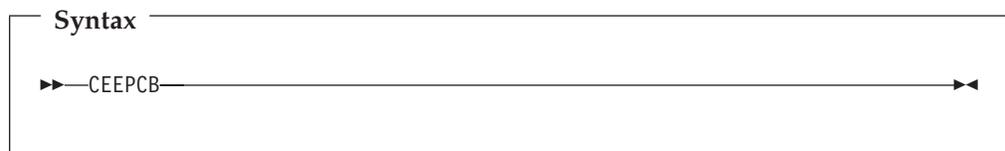
name The optional name operand, which becomes the label on the exit from this routine. The name does not have to match the prolog.

CEERCB Macro — Generate an RCB Mapping



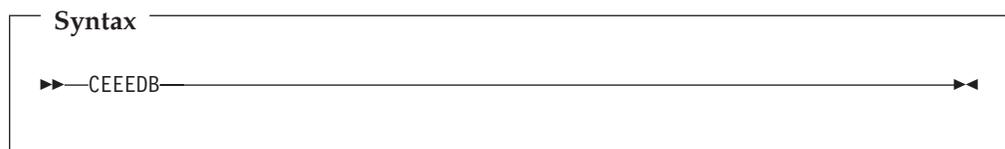
CEERCB is used to generate a region control block (RCB) mapping. This macro has no parameters, and no label can be specified.

CEEPCB Macro — Generate an PCB Mapping



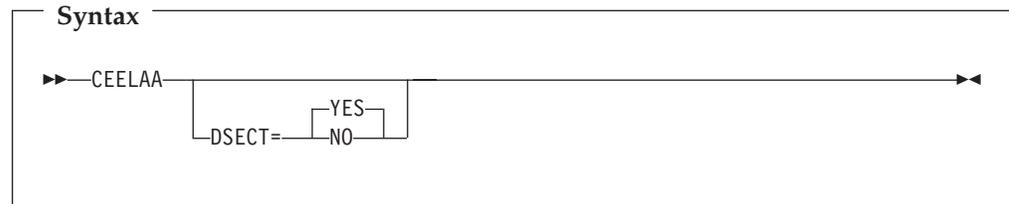
CEEPCB is used to generate a process control block (PCB) mapping. This macro has no parameters, and no label can be specified.

CEEEDB Macro — Generate an EDB Mapping



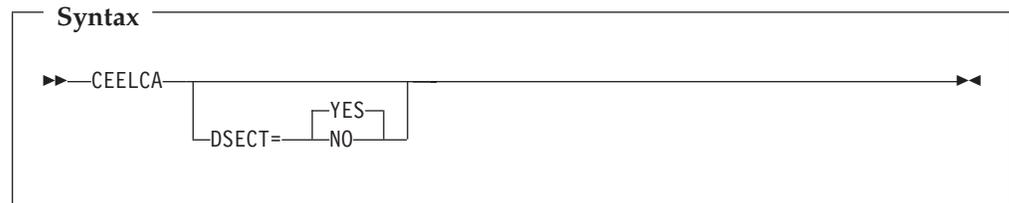
CEEEDB is used to generate an enclave data block (EDB) mapping. This macro has no parameters, and no label can be specified.

CEELAA Macro — Generate an LAA Mapping



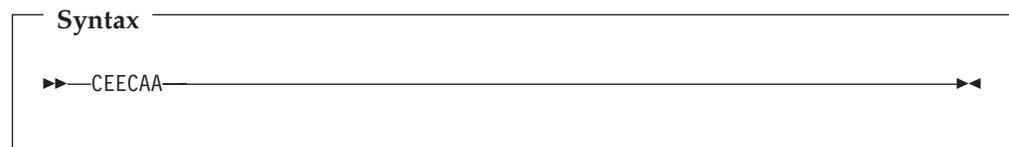
CEELAA is used to generate a library anchor area (LAA) mapping. If the optional DSECT parameter is specified as YES (which is also the default), then a DSECT definition of the LAA will be produced, otherwise storage will be reserved (with labels) for the LAA.

CEELCA Macro — Generate an LCA Mapping



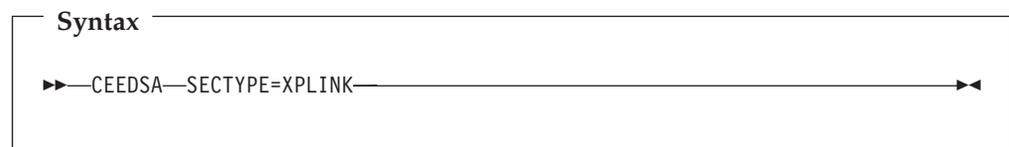
CEELCA is used to generate a library communication area (LCA) mapping. If the optional DSECT parameter is specified as YES (which is also the default), then a DSECT definition of the LCA will be produced, otherwise storage will be reserved (with labels) for the LCA.

CEECAA Macro — Generate a CAA Mapping



CEECAA is used to generate a common anchor area (CAA) mapping. This macro has no parameters, and no label can be specified. CEECAA is required for the CEEENTRY macro.

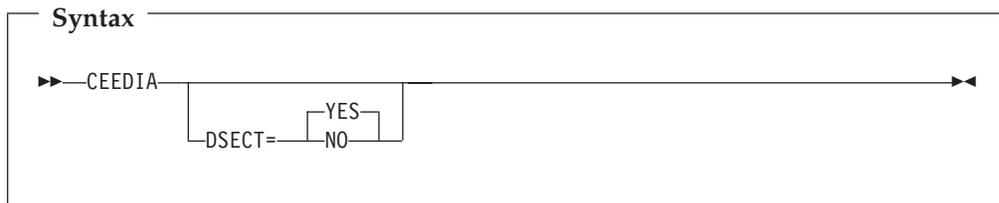
CEEDSA Macro — Generate a DSA Mapping



Assembler Considerations

CEEDSA is used to generate a dynamic save area (DSA) mapping. The parameter SECTYPE=XPLINK must be specified in order to generate a 64-bit XPLINK DSA. The minimum size of a 64-bit XPLINK DSA is contained in an assembler EQUATE CEEDSAHPSZ.

CEEDIA Macro — Generate an DIA Mapping



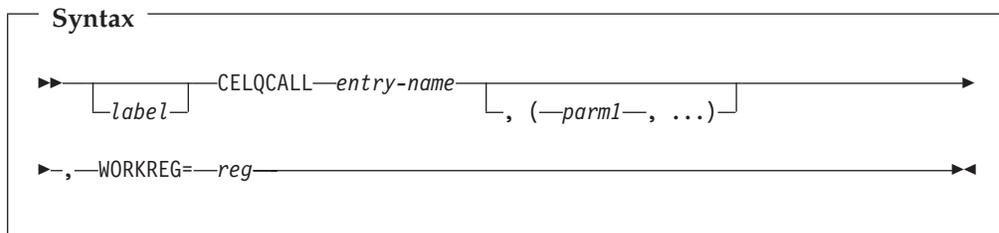
CEEDIA is used to generate a debugger interfaces area (DIA) mapping. If the optional DSECT parameter is specified as YES (which is also the default), then a DSECT definition of the DIA will be produced, otherwise storage will be reserved (with labels) for the DIA.

CELQCALL Macro — Call a Language Environment-Conforming AMODE 64 Routine

CELQCALL can be used to pass control from an AMODE 64 assembler program to another AMODE 64 control section at a specified entry point. It is meant to be used in conjunction with the CELQPRLG prolog and CELQEPLG macros. The target of CELQCALL can be resolved either statically (bound with the same program object) or dynamically (imported from a DLL).

The CELQCALL macro does not generate any return codes. Return information may be placed in GPR 3 (and possibly GPRs 2 and 1, or the Floating Point Registers) by the called program, as specified by 64-bit XPLINK linkage conventions. For more information, refer to *z/OS Language Environment Vendor Interfaces*.

GPRs 0, 1, 2, 3, 5, 6, and 7 are not preserved by this macro.



label Optional label beginning in column 1.

entry-name Specifies the entry name of the program to be given control. This entry name can reside in the same program object, or can be an exported DLL function.

, (parm1, ...)

One or more parameters to be passed to the called program. The parameters are copied to the argument area in the calling

program's DSA, and then GPRs 1, 2, and 3 are loaded with the first three words of this argument area. Sufficient space must be reserved in the caller's argument area to contain the largest possible parameter list. A minimum of 8 words (32 bytes) must always be allocated for the argument area. Use the DSASIZE= parameter on the CELQPRLG prolog macro to ensure the calling program's DSA is large enough.

At this time, the CELQCALL macro only supports passing parameters by reference.

WORKREG= A numeric value representing a general purpose register between 8 and 15, inclusive, that can be used as a work register by this macro. Its contents will not be preserved.

Usage Notes:

1. This macro requires that the calling routine's 64-bit XPLINK environment address is in register 5 (as it was when the routine was first invoked).
2. This macro requires that a PSECT was defined by the CELQPRLG prolog macro.
3. This macro requires the GOFF assembler option.
4. This macro requires the binder to bind, and the RENT and DYNAM(DLL) binder options. You will also need the CASE(MIXED) binder option if the entry-name is mixed case.
5. The output from the binder must be a PM4 (or higher) format program object, and therefore must reside in either a PDSE or the HFS.

The following AMODE 64 assembler example shows a call to an AMODE 64 routine named Xif1 where no parameters are passed.

```
ADLA6IF1 CELQPRLG DSASIZE=DSASZ,PSECT=ADLA6IFP
*
R3      EQU   3          RETURN VALUE
*
      WTO   'ADLA6IF1: Calling imported AMODE 64 function Xif1',   X
      ROUNTCDE=11
*
      CELQCALL Xif1,WORKREG=10
*
      SGR   R3,R3
RETURN  DS    0H
      CELQEPLG
*
      LTORG
CEEDSAHP CEEDSA SECTYPE=XPLINK
MINARGA  DS    8F
DSASZ    EQU  *-CEEDSAHP_FIXED
      END    ADLA6IF1
```

This is an example of AMODE 64 assembler code calling a function with 5 parameters.

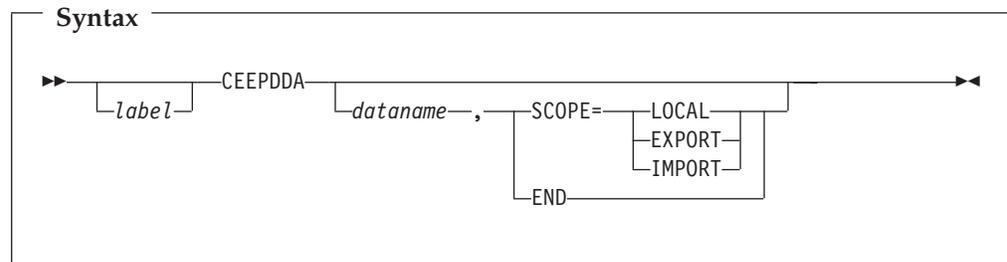
Assembler Considerations

```
ADLA6IF7 CELQPRLG DSASIZE=DSASZ,PSECT=ADLA6IFP
*
R3      EQU   3          RETURN VALUE
*
      WTO   'ADLA6IF7: Calling imported AMODE 64 function Xif7 passiX
            ng parmeters (15,33,"Hello world",45.2,9)',          X
            ROUTCDE=11
*
      CELQCALL Xif7,(PARM1,PARM2,PARM3,PARM4,PARM5),WORKREG=10
*
      SGR   R3,R3
RETURN  DS    0H
      CELQEPLG
*
      LTOrg
PARM1   DC   FL4'15'
PARM2   DC   FL2'33'
PARM3   DC   C'Hello world'
      DC   X'00'
PARM4   DC   D'45.2'
PARM5   DC   FL4'9'
CEEDSAHP CEEDSA SECTYPE=XPLINK
ARGAREA DS    10F
DSASZ   EQU  *-CEEDSAHP_FIXED
      END    ADLA6IF7
```

CEEPDDA Macro — Define a Data Item in the Writeable Static Area (WSA)

CEEPDDA can be used to define data in WSA, and optionally specify it as either exported or imported data.

If the CEEPDDA macro is followed by data constant(s), it is declared data, and must be followed by a subsequent CEEPDDA invocation with only the END parameter to mark the end of the declared data. If there are no subsequent data constants, a reference will be created for the imported data.



label Optional label beginning in column 1.

dataname Specifies the name of the data item. It is case sensitive and can be up to 255 characters in length. This name can reside in the same program object, or can be an exported DLL function.

SCOPE= {LOCAL|EXPORT|IMPORT}

Optional keyword parameter that results in the data being exported if SCOPE=EXPORT is specified and this instance of CEEPDDA is to declare data, or the data being imported if

SCOPE=IMPORT is specified and this instance of CEEPDDA generates a reference to data (i.e. no data constants follow macro). The use of SCOPE=LOCAL can be used to declare data in WSA that is not exported.

END The use of CEEPDDA with the END parameter is used to indicate the end of this defined data item, and must be used in conjunction with an invocation of CEEPDDA with the SCOPE=EXPORT or SCOPE=LOCAL keyword parameter.

Usage Notes:

1. This macro requires the GOFF assembler option.
2. This macro requires the binder to bind, and the RENT and DYNAM(DLL) binder options. You will also need the CASE(MIXED) binder option if the *dataname* is mixed case.
3. The output from the binder must be a PM4 (or higher) format program object, and therefore must reside in either a PDSE or the HFS.

For more details on DLLs, including full sample assembler DLL routines, see 13.

The following example illustrates how to export data from assembler. The first exported data item is an integer with the initial value 123, and the second exported data item is the character string "Hello World" with a terminating NULL (x'00') character:

```
CEEPDDA D11Var,SCOPE=EXPORT
DC      A(123)
CEEPDDA END
CEEPDDA D11Str,SCOPE=EXPORT
DC      C'Hello World'
DC      X'00'
CEEPDDA END
```

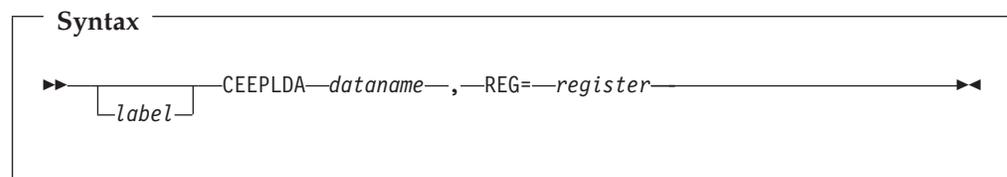
The following example illustrates how to import the variable named Biv1 into assembler.

```
CEEPDDA Biv1,SCOPE=IMPORT
```

CEEPLDA Macro — Returns the Address of a Data Item Defined by CEEPDDA

CEEPLDA is used to obtain the address of a local, imported, or exported data item. The required *dataname* label will name the data item, is case sensitive, and can be up to 255 characters in length.

Registers 0, 14, and 15 are not preserved by this macro.



label Optional label beginning in column 1.

dataname Specifies the name of the data item whose address will be returned. It is case sensitive and can be up to 255 characters in length.

Assembler Considerations

REG= The numeric value of the register to contain the address of the data identified by *dataname*. Registers 0, 14, and 15 cannot be used.

Usage Notes:

1. This macro requires the GOFF assembler option.
2. This macro requires the binder to bind, and the RENT and DYNAM(DLL) binder options. You will also need the CASE(MIXED) binder option if the *dataname* is mixed case.
3. The output from the binder must be a PM3 (or higher) format program object, and therefore must reside in either a PDSE or the HFS.

For more details on DLLs, including full sample assembler DLL routines, see 13.

The following example illustrates how to obtain the address of an imported variable in WSA and store an integer value into it. This particular example uses a corresponding CEEPDDA instance for an imported variable, but an exported or local variable would also work (see “CEEPDDA Macro — Define a Data Item in the Writeable Static Area (WSA)” on page 158).

```
* Obtain address of imported variable Biv1 in register 9
   CEEPLDA Biv1,REG=9
* Set value of imported variable to 123
   LA     R8,123
   ST     R8,0(,R9)
...
   CEEPDDA Biv1,SCOPE=IMPORT
```

Chapter 22. Using preinitialization services with AMODE 64

Language Environment Preinitialization (PreInit) is commonly used to enhance performance for repeated invocations of an application or for a complex application where there are many repetitive requests and where fast response is required. For instance, if an assembler routine invokes either a number of Language Environment-conforming HLL routines or the same HLL routine a number of times, the creation and termination of that HLL environment multiple times is needlessly inefficient. A more efficient method is to create the HLL environment only once for use by all invocations of the routine.

PreInit lets an application initialize an HLL environment once, perform multiple executions using that environment, and then explicitly terminate the environment. Because the environment is initialized only once (even if you perform multiple executions), you free up system resources and allow for faster responses to your requests.

In the 64-bit environment, CELQPIPI, will provide the interface for preinitialized routines. Using CELQPIPI, you can initialize an environment, invoke applications, terminate an environment, and add an entry to the PreInit table. (The PreInit table contains the names and entry point addresses of routines that can be executed in the preinitialized environment.)

CEEPIPI will continue to be the 31-bit PreInit interface.

Understanding the basics

PreInit support can be divided into two areas:

- PreInit assembler driver support

Users must create a PreInit driver that is used to create a PreInit environment. The driver will setup the PreInit environment, load the PreInit interface CELQPIPI. CELQPIPI is loaded RMODE(31), AMODE(64). All PreInit requests must be made from a non-Language Environment-conforming driver (such as assembler).

The following assembler macros are provided to be used in the driver to define the PreInit table:

- CELQPIT PreInit table header
- CELQPITY PreInit table entry
- CELQPITS PreInit table end

The driver will be used to invoke the PreInit interface by using function calls to CELQPIPI using the standard OS linkage parameter list.

Function calls to CELQPIPI from the customer PreInit driver:

- init_main
- init_sub
- call_main
- call_sub
- call_sub_addr
- term
- start_seq

- end_seq
- add_entry
- delete_entry
- identify_entry
- identify_attributes
- Application program support running in the PreInit environment.
The PreInit table contains the names and entry point addresses of each routine that can be executed within the PreInit environment.
The applications defined in the PreInit table must be able to run as AMODE 64 (with XPLINK implied).
Languages Supported:
 - C
 - C++
 - Assembler (64-bit Language Environment-conforming assembler)
 - Language mix of the above

Using preinitialization

Language Environment Preinitialization consists of three parts:

- PreInit Table
- Assembler Driver Program
- CELQPIPI Service

The PreInit Table is built by the user (using Language Environment supplied macros). It identifies the set of routines that the user will be invoking within a given pre-initialized environment. Language Environment uses this table to pre-load these routines during PreInit initialization and to identify the set of Language Environment languages that may be used by this pre-initialized environment.

The assembler Driver program is a user-written program that is used to control a pre-initialized environment. It consists of a series of calls to the CELQPIPI service, to initialize a PreInit environment, invoke user routines, and terminate a PreInit environment.

The CELQPIPI is a service provided by Language Environment to support 64-bit pre-initialized environments. All of the operations that are required to use a PreInit environment are accessed through calls to this service.

Macros that generate the Preinit table

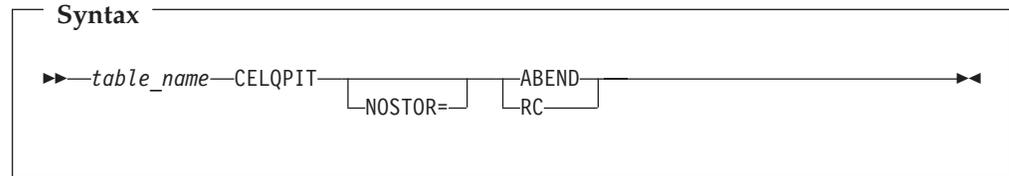
Language Environment provides the following assembler macros to generate the PreInit table for you:

- CELQPIT
- CELQPITY
- CELQPITS

These macro definitions can be found in the Language Environment SCEEMAC data set, which should be specified on the SYSLIB DD statement when assembling your program.

CELQPIT

CELQPIT generates a header for the PreInit table.



table_name

Assembler symbolic name assigned to the first word in the PreInit table. The address of this symbol should be used as the *ceexptbl_addr* parameter in a CELQPIPI(*init_main*) or a CELQPIPI(*init_sub*) call.

NOSTOR=

Indicates whether CELQPIPI caller wants Language Environment to abend or return a return code if initial storage allocations are not successful.

NOSTOR=ABEND

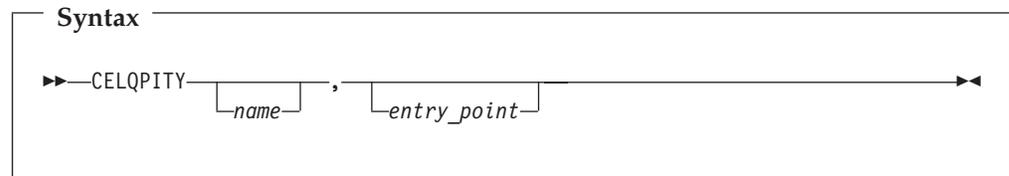
Indicates abend is desired (default)

NOSTOR=RC

Indicates a return code is desired

CELQPITY

CELQPITY generates an entry within the PreInit table.



name

An eight character string containing the load name of the routine that can be invoked within the Language Environment preinitialized environment. This name must be provided if Language Environment is to load the routine, otherwise it is optional.

entry_point

Doubleword routine address to which control is transferred, or 0, to indicate that the module is to be dynamically loaded. This parameter is optional.

You have the option of specifying either, both, or neither of these parameters:

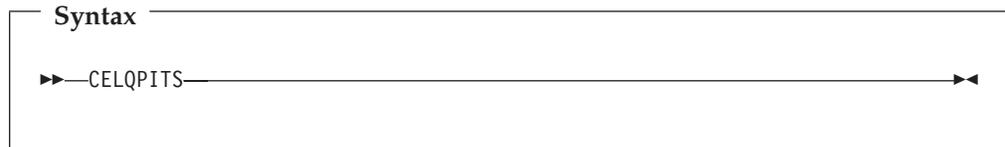
- If *name* is omitted and *entry_point* is present, the comma must be present.
- If neither *name* nor *entry_point* are provided, an empty PIT entry is created which can be filled in later with the CELQPIPI(*add_entry*) preinitialization function.
- If *name* is provided and *entry_point* is zero, Language Environment will load the routine.
- If both parameters are present, *name* is ignored and *entry_point* is used as the start of the routine, unless it is 0.

Note: Each invocation of the CELQPITY macro generates a new row in the PreInit table. The first entry is row 0, the second is row 1, and so on.

CELQPITS

CELQPITS identifies the end of the PreInit table.

This macro has no parameters.



Invoking CELQPIPI

The interface to the PreInit environment is from a user-written AMODE 64 PreInit driver. The CELQPIPI service is loaded using the LOAD macro:

```
LOAD EP=CELQPIPI
```

The entry point address returned in register 0 by the LOAD macro will have bit 63 set to 1, which indicates that CELQPIPI is AMODE 64. Bit 63 must be set to 0 before the address can be used by either the CALL macro or a BASR instruction.

AMODE Considerations

CELQPIPI does not perform any AMODE switching. CELQPIPI expects to receive control in AMODE 64. It always returns control in the same AMODE in which it received control. If, on entry, the AMODE is not 64, CELQPIPI will return with return code 48.

General Register Usage at Entry to CELQPIPI

The following registers must have the prescribed contents when control reaches CELQPIPI.

- R0** Undefined.
- R1** Must point to an indirect parameter list consisting of 8 byte addresses that point to the parameters.
- R2-R12** Undefined.
- R13** Must point to a 36-word doubleword-aligned format 4 save area. See *z/OS MVS Programming: Assembler Services Guide* for a description of a format 4 save area.
- R14** The return address.
- R15** The address of CELQPIPI.

General Register Usage at Exit from CELQPIPI

Registers have the following contents when control returns to the caller of CELQPIPI.

- R0** Not preserved.
- R1** Not preserved.
- R2-R12** Preserved.

- R13 Preserved.
- R14 Not preserved.
- R15 Return code.

CELQPIPI interface

The following section describes how to invoke the PreInit interface, CELQPIPI, to perform the following tasks:

- Initialization
- Application invocation
- Termination
- Addition of an entry to the PreInit table
- Deletion of a main entry from the PreInit table
- Identification of an entry in the PreInit table

The PreInit services offered under Language Environment, using CELQPIPI are listed in Table 25.

Table 25. Preinitialization Services Accessed Using CELQPIPI

Function Code	Integer Value	Service Performed
Initialization		
<i>init_main</i>	1	Create and initialize an environment for multiple executions of main routines.
<i>init_sub</i>	3	Create and initialize an environment for multiple executions of subroutines.
Application invocation		
<i>call_main</i>	2	Invoke a main routine within an already initialized environment.
<i>call_sub</i>	4	Invoke a subroutine within an already initialized environment.
<i>call_sub_addr</i>	10	Invoke a subroutine by function descriptor.
Termination		
<i>term</i>	5	Explicitly terminate the environment without executing a user routine.
Invoke a Sequence of Applications		
<i>start_seq</i>	7	Start a sequence of uninterruptable calls to a number of subroutines.
<i>end_seq</i>	8	Terminate a sequence of uninterruptable calls to a number of subroutines.
Addition of an entry to PreInit table		
<i>add_entry</i>	6	Dynamically add a candidate routine to execute within the preinitialized environment.
Deletion of an entry from PreInit table		
<i>delete_entry</i>	11	Delete an entry from the PreInit table, making it available for subsequent <i>add_entry</i> functions.
Identification of a PreInit table entry		
<i>identify_entry</i>	13	Identify the programming language of an entry in the PreInit table.
<i>identify_attributes</i>	16	Identify the attributes of an entry in the PreInit table.

Initialization

Language Environment supports two forms of preinitialized environments. The first, supports the execution of main routines. The second, supports the execution of subroutines.

The primary difference between these environments is the amount of Language Environment initialization (and termination) that occurs on each application invocation call. With an environment that supports main routines, most of the application's execution environment is reinitialized with each invocation. With an environment that supports subroutines, very little of the execution environment is reinitialized with each invocation. This difference has its advantages and disadvantages.

- **Main environment:**
 - **Advantages:**
 - A new, pristine environment is created.
 - Run-time options can be specified for each application.
 - **Disadvantages:**
 - Poorer performance.
- **Sub environment:**
 - **Advantages:**
 - Best performance.
 - **Disadvantages:**
 - The environment is left in what ever state the previous application left it in.
 - Run-time options cannot be changed.

CELQPIPI(*init_main*) — **initialize for main routines:** The invocation of this routine:

- Creates and initializes a new common run-time environment (process) that allows the execution of main routines multiple times
- Sets the environment dormant so that exceptions are percolated out of it
- Returns a token identifying the environment to the caller
- Returns a code in Register 15 indicating whether an environment was successfully initialized

Syntax

```
▶▶—CALL—CELQPIPI—(—init_main—,—ceexptbl_addr—,——————▶  
▶—service_rtns—,—token—)——————▶▶
```

init_main (input)

A fullword function code (integer value = 1) containing the *init_main* request.

ceexptbl_addr (input)

A doubleword containing the address of the PreInit table to be used during initialization of the new environment. Language Environment does not alter the user-supplied copy of the table. If an entry address is zero and the entry name is non-blank, Language Environment searches for the routine and

dynamically loads it. Language Environment places the entry address in the corresponding slot of a Language Environment-maintained table.

service_rtns (input)

A 64-bit pointer containing the address of the service routine vector or 0, if there is no service routine vector. See “Service Routines” on page 179 for more information.

token (output)

A doubleword containing a unique value used to represent the environment.

The *token* should be used only as input to additional calls to CELQPIPI, and should not be altered or used in any other manner.

Return Codes: Register 15 contains a return code indicating if an environment was successfully initialized. Possible return codes are:

- 0 A new environment was successfully initialized.
- 4 The function code is not valid.
- 8 All addresses in the table were not resolved. One or more LOAD failures were encountered. Initialization continues.
- 12 Storage for the preinitialization environment could not be obtained.
- 32 All addresses in the table were not resolved. One or more routines within the table were not AMODE 64 or were generated by a non Language Environment conforming HLL. Initialization continues.
- 40 All addresses in the table were not resolved. One or more LOAD failures were encountered, and one or more routines within the table were not AMODE 64 or were generated by a non Language Environment conforming HLL. Initialization continues.
- 48 CELQPIPI was called from a non-64-bit environment.
- 56 An unhandled condition occurred.
- 60 One or more reserved fields in the service routine vector were non-zero.
- 64 In the service routine vector, the @LOAD field was non-zero and the @DELETE field was zero, or the @LOAD field was zero and the @DELETE field was non-zero. The LOAD and DELETE routines must both be present in the service vector, or both fields must be zero.
- 68 In the service routine vector, the @GETSTORE field was non-zero and the @FREESTORE field was zero, or the @GETSTORE field was zero and the @FREESTORE field was non-zero. The GETSTORE and FREESTORE routines must both be present in the service vector, or both fields must be zero.

Usage Notes:

- The identify_attributes function can be used to determine what entry in the PreInit table failed to load when a return code of 8 was returned.
- CELQPIPI supports the creation of multiple PreInit Main environments.
 - C/C++ main routines must be initialized with (init_main).
 - C/C++ routines that are the target of (call_main) must contain a main().

Restriction:

- Only one PreInit environment, per TCB, may have the POSIX runtime option set to ON.

CELQPIPI(init_sub) — initialize for subroutines: The invocation of this routine:

- Creates and initializes a new common run-time environment (process and enclave) that allows the execution of subroutines multiple times
- Sets the environment dormant so that exceptions are percolated out of it

- Returns a token identifying the environment to the caller
- Returns a code in Register 15 indicating whether an environment was successfully initialized
- Ensures that when the environment is dormant, it is immune to other Language Environment enclaves that are created or terminated

Syntax

```

▶▶—CALL—CEELQPIPI—(—init_sub—,—ceexptbl_addr—,——————▶
▶—service_rtns—,—runtime_opts—,—token—)——————▶▶

```

init_sub (input)

A fullword function code (integer value = 3) containing the *init_sub* request.

ceexptbl_addr (input)

A doubleword containing the address of the PreInit table to be used during initialization of the new environment. Language Environment does not alter the user-supplied copy of the table. If an entry address is zero and the entry name is nonblank, Language Environment searches for the routine and dynamically loads it. Language Environment then places the entry address in the corresponding slot of Language Environment's copy of the PreInit table.

service_rtns (input)

A 64-bit pointer containing the address of the service routine vector or 0, if there is no service routine vector. See "Service Routines" on page 179 for more information.

runtime_opts (input)

A fixed-length 255-character string containing runtime options (see *z/OS Language Environment Programming Reference* for a list of runtime options that you can specify).

Note: The runtime options you specify will apply to all of the subroutines that are called by the (*call_sub*) function. This includes options such as POSIX. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.

token (output)

A doubleword containing a unique value used to represent the environment. The *token* should be used only as input to additional calls to CELQPIPI, and should not be altered or used in any other manner.

Return Codes: Register 15 contains a return code indicating the success or failure of the call. Possible return codes are:

- | | |
|----|---|
| 0 | A new environment was successfully initialized. |
| 4 | The function code is not valid. |
| 8 | All addresses in the table were not resolved. One or more LOAD failures were encountered. Initialization continues |
| 12 | Storage for the preinitialization environment could not be obtained. |
| 32 | All addresses in the table were not resolved. One or more routines within the table were not AMODE 64 or were not fetchable or were generated by a non Language Environment conforming HLL. Initialization continues. |
| 40 | All addresses in the table were not resolved. One or more LOAD failures were encountered, and one or more routines within the table were not |

- AMODE 64 or were not fetchable or were generated by a non Language Environment conforming HLL . Initialization continues.
- 48 CELQPIPI was called from a non-64-bit environment.
- 56 An unhandled condition occurred.
- 60 One or more reserved fields in the service routine vector were non-zero.
- 64 In the service routine vector, the @LOAD field was non-zero and the @DELETE field was zero, or the @LOAD field was zero and the @DELETE field was non-zero. The LOAD and DELETE routines must both be present in the service vector, or both fields must be zero.
- 68 In the service routine vector, the @GETSTORE field was non-zero and the @FREESTORE field was zero, or the @GETSTORE field was zero and the @FREESTORE field was non-zero. The GETSTORE and FREESTORE routines must both be present in the service vector, or both fields must be zero.

Usage Notes:

- The identify_attributes function can be used to determine what entry in the PreInit table failed to load when a return code of 8 was returned.
- CELQPIPI supports the creation of multiple PreInit Subroutine environments.

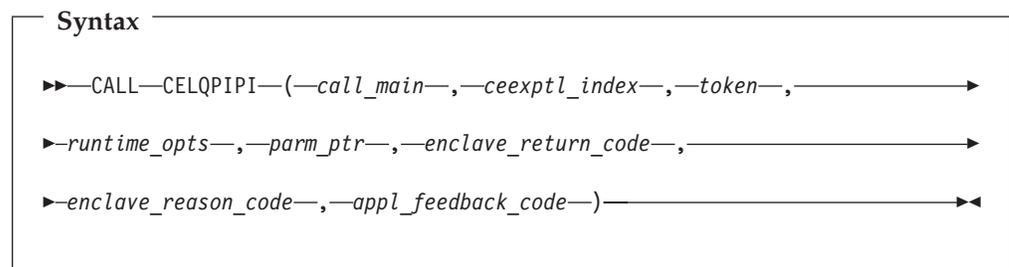
Restriction:

- Only one PreInit environment, per TCB, may have the POSIX runtime option set to ON.

Application invocation

Language Environment provides multiple facilities for invoking either main routines or subroutines. When invoking a main routine, the preinitialized environment must have been created and initialized using the CELQPIPI(init_sub) function. Similarly, when invoking a subroutine, the preinitialized environment must have been created and initialized using the CELQPIPI(init_sub) function.

CELQPIPI(call_main) — invocation for main routine: This invocation of CELQPIPI invokes as a main routine the routine that you specify.



call_main (input)

A fullword function code (integer value = 2) containing the call_main request.

ceexptbl_index (input)

A doubleword containing the row number within the PreInit table of the entry that should be invoked. The index starts at 0.

token (input)

A doubleword with the value of the token returned by CELQPIPI(init_main) when the common run-time environment is initialized. The token must identify a previously preinitialized environment that is not active at the time of the call.

runtime_opts (input)

A fixed-length 255-character string containing runtime options. (See *z/OS Language Environment Programming Reference* for a list of runtime options that you can specify.)

parm_ptr (input)

A doubleword containing the address of the parameter list or 0 (zero). The parameter list is copied to the appropriate location in the stack frame and general purpose registers 1, 2 and 3 are loaded from the parameter list when the main routine is executed. The parameter list that is passed must be in a format that HLL subroutines expect (for example, in an argc, argv format for C routines).

enclave_return_code (output)

A fullword containing the enclave return code returned by the called routine when it finished executing. For more information about return codes, see "Managing return codes in Language Environment" on page 69.

enclave_reason_code (output)

A fullword containing the enclave reason code returned by the environment when the routine finished executing. For more information about reason codes, see "Managing return codes in Language Environment" on page 69.

appl_feedback_code (output)

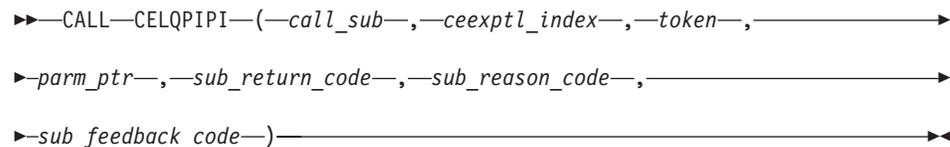
A 128-bit condition token indicating why the application terminated.

Return Codes: A return code is provided in register 15 and can contain the following values:

- 0 The environment was activated and the routine called.
- 4 The function code is not valid.
- 12 The indicated environment was initialized for subroutines. No routine was executed.
- 16 The *token* is not valid.
- 20 The index points to an entry that is not valid or empty.
- 24 The index that was passed is outside the range of the table.
- 48 CELQPIPI was called from a non-64-bit environment.

CELQPIPI(call_sub) — invocation for subroutines: This invocation of CELQPIPI invokes as a subroutine the routine that you specify. The environment is not reinitialized.

Syntax



call_sub (input)

A fullword function code (integer value = 4) containing the call_sub request for a subroutine.

ceexptl_index (input)

A doubleword containing the row number of the entry within the PreInit table that should be invoked; the index starts at 0.

***token* (input)**

A doubleword with the value of the token returned when the common run-time environment is initialized. This token is initialized by the (init_sub). The *token* must identify a previously preinitialized environment that is not active at the time of the call.

***parm_ptr* (input)**

A doubleword containing the address of the parameter list or 0 (zero). The parameter list is copied to the appropriate location in the stack frame and general purpose registers 1, 2 and 3 are loaded from the parameter list when the routine is executed.

The length of the parameter list is determined from PPAs of the routine. If the routine takes a variable length parameter list, the length of the parameter list will be assumed to be 256 bytes

Floating point and complex values can only be passed by reference.

***sub_ret_code* (output)**

A fullword containing the subroutine return code. When a PreInit call_sub ends normally, the sub_ret_code is set with the value of R15 returned from the subroutine. If the enclave is terminated due to an error or due to the routine explicitly invoking a language-specific service that causes a stop or exit semantic, this contains the enclave return code for termination.

***sub_reason_code* (output)**

A fullword containing the subroutine reason code. This is 0 for normal subroutine returns. If the enclave is terminated due to an error or due to the routine explicitly invoking a language-specific service that causes a stop or exit semantic, this contains the enclave reason code for termination.

***sub_feedback_code* (output)**

A 16-byte field containing the feedback code for enclave termination. This is the CEE000 feedback code for normal subroutine returns. If the enclave is terminated due to an error or due to the routine explicitly invoking a language-specific service that causes a stop or exit semantic, this contains the enclave feedback code for termination.

Return Codes: A return code is provided in register 15 and can contain the following values:

- 0 The environment was activated and the routine called.
- 4 The function code is not valid.
- 12 The indicated environment was initialized for main routines. No routine was executed.
- 16 The *token* is not valid.
- 20 The index points to an entry that is not valid or empty.
- 24 The index passed is outside the range of the table.
- 28 The enclave was terminated but the process level persists.

This value indicates the enclave was terminated while the process was retained. This can occur due to a STOP statement being issued or due to an unhandled condition. The *sub_ret_code*, *sub_reason_code*, and *sub_feedback_code* indicate this action.

- 48 CELQPIPI was called from a non-64-bit environment.

CELQPIPI(call_sub_addr) — invocation for subroutines by address: This PreInit call is the same as 'call_sub' except the routine is identified by its address, not its index in the PreInit table.

Syntax

```
▶▶CALL—CELQPIPI—(—call_sub_addr—,—routine_addr—,——————▶
▶function_pointer—,—token—,—parm_ptr—,—sub_ret_code—,—————▶
▶sub_reason_code—,—sub_feedback_code—)—————▶▶
```

call_sub_addr (input)

A fullword function code (integer value = 10) containing the call_sub request for a subroutine.

routine_addr (input/output)

A doubleword containing the address of the routine that should be invoked. If this value is zero, then the function_pointer will be used to invoke the routine.

function_pointer (input/output)

A 16 byte field used to invoke the routine directly. The first time (and first time after enclave termination occurred) a routine is called, this field must be zero. On subsequent calls to the same routine, the value returned in this field should be used. This will ensure that there is only one copy of writable static area for the routine and that static constructors are only run once.

token (input)

A doubleword with the value of the token returned by (init_sub) when the common run-time environment is initialized.

The *token* must identify a previously preinitialized environment that is not active at the time of the call.

Note: If the token pointing to the previously preinitialized environment is a non-XPLINK environment and the subprogram to be invoked is XPLINK, then a return code of 40 will be returned because this is not valid.

parm_ptr (input)

A doubleword containing the address of the parameter list or 0 (zero). The parameter list is copied to the appropriate location in the stack frame and general purpose registers 1, 2 and 3 are loaded from the parameter list when the routine is executed.

The length of the parameter list is determined from PPAs of the routine. If the routine takes a variable length parameter list, the length of the parameter list will be assumed to be 256 bytes

Floating point and complex values can only be passed by reference.

sub_ret_code (output)

A fullword containing the subroutine return code. When a PreInit call_sub ends normally, the sub_ret_code is set with the value of R15 returned from the subroutine. If the enclave is terminated due to an error or due to the routine explicitly invoking a language-specific service that causes a stop or exit semantic, this contains the enclave return code for termination.

sub_reason_code (output)

A fullword containing the subroutine return code. If the enclave is terminated

due to an error or due to the routine explicitly invoking a language-specific service that causes a stop or exit semantic, this contains the enclave return code for termination.

sub_feedback_code (output)

A 16-byte field containing the feedback code for enclave termination. This is the CEE000 feedback code for normal subroutine returns. If the enclave is terminated due to an error or due to the routine explicitly invoking a language-specific service that causes a stop or exit semantic, this contains the enclave feedback code for termination.

Return Codes: A return code is provided in register 15 and can contain the following values:

- 0 The environment was activated and the routine called.
- 4 The function code is not valid.
- 12 The indicated environment was initialized for main routines. No routine was executed.
- 16 The *token* is not valid.
- 28 The enclave was terminated but the process level persists.

This value indicates the enclave was terminated while the process was retained. This can occur due to a STOP or EXIT statement (or an exit() function) being issued or due to an unhandled condition. The *sub_ret_code*, *sub_reason_code*, and *sub_feedback_code* indicate this action.

- 32 The *function_pointer* passed is not a valid function pointer.
- 48 CELQPIPI was called from a non-64-bit environment.
- 52 The *function_pointer* could not be obtained from the *routine_addr*.

Invocation of a sequence of applications: When the driver program needs to invoke multiple subroutines in a sequence, improved performance can be obtained through the use of the *start_seq* and *end_seq* PreInit functions. The facility will minimize the overhead of each program invocation.

In order to exploit this feature the user must ensure that no activity is performed by the driver program between the PreInit calls, and all of these calls must be to the same preinitialized environment.

To use this facility the driver program first calls CELQPIPI(*start_seq*), then is can make multiple calls to CELQPIPI(*call_sub*) or CELQPIPI(*call_sub_addr*), and finally it ends the sequence with a call to CELQPIPI(*end_seq*).

CELQPIPI(start_seq) — *start a sequence of calls:* This invocation of CELQPIPI declares that a sequence of uninterrupted calls is made to a number of subroutines by this driven program to the same preinitialized environment. This minimizes the overhead between calls by performing as much activity as possible at the start of a sequence of calls.

Syntax

▶▶ CALL CELQPIPI (—*start_seq*—, —*token*—) ▶▶

start_seq (input)

A fullword function code (integer value = 7) containing the *start_seq* request.

token (input)

A doubleword with the value of the token returned by CELQPIPI(init_sub) when the common run-time environment is initialized.

The *token* must identify a previously preinitialized environment for subroutines that are dormant at the time of the call.

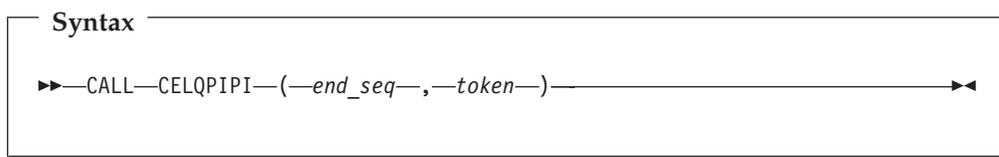
Return Codes: A return code is provided in register 15 and can contain the following values:

- 0 The environment was prepared for a sequence of calls.
- 4 The function code is not valid.
- 16 The *token* is not valid.
- 20 Sequence already started using *token*.
- 48 CELQPIPI was called from a non-64-bit environment.

Usage Notes:

- CELQPIPI(start_seq) may only be used in conjunction with preinitialized environment that was created and initialized by CELQPIPI(init_sub). A return code of 4 is set if the environment was not created by CELQPIPI(init_sub), since this function code is invalid for a token that is not for a subroutine.
- CELQPIPI(start_seq) minimizes the overhead of invoking subroutines by performing as much activity as possible at the start of the sequence of calls, thus reducing the activity performed on each subroutine invocation within the sequence.
- Only CELQPIPI(call_sub) and CELQPIPI(call_sub_addr) invocation are allowed between the CELQPIPI(start_seq) and the CELQPIPI(end_seq) calls.
- The same PreInit preinitialized environment token must be used on all of the PreInit calls within a sequence, including the calls to CELQPIPI(start_seq) and CELQPIPI(end_seq). Otherwise return code is set for invalid token (16).

CELQPIPI(end_seq) — end a sequence of calls: This invocation of CELQPIPI declares that a sequence of uninterrupted calls to subroutines by this driver program has finished.



end_seq (input)

A fullword function code (integer value = 8) containing the end_seq request

token (input)

A doubleword with the value of the token returned by CELQPIPI(init_sub) when the common run-time environment is initialized.

The *token* must identify a previously preinitialized environment for subroutines that are dormant at the time of the call.

Return Codes: A return code is provided in register 15 and can contain the following values:

- 0 The environment is no longer prepared for a sequence of calls.
- 4 The function code is not valid.
- 16 The *token* is not valid.
- 20 The *token* was not used in a start_seq call.

48 CELQPIPI was called from a non-64-bit environment.

PreInit termination: Although there are two types of preinitialized environments, and two functions to create them, there is only one function to terminate these environments.

The call to CELQPIPI(*term*) will terminate a single preinitialized environment, and it is the responsibility of the driver program to issue the *term* function for each preinitialized environment created.

CELQPIPI(term) — *terminate environment*: This invocation of CELQPIPI terminates the environment identified by the value given in *token*. This service is used for terminating environments created for subroutines or main routines.

Syntax

```
▶▶ CALL CELQPIPI (—term—, —token—, —env_return_code—) ▶▶
```

term (input)

A fullword function code (integer value = 5) containing the termination request.

token (input)

A doubleword with the value of the token of the environment to be terminated. This token is returned by a CELQPIPI(*init_main*), or CELQPIPI(*init_sub*) request during the initialization call.

The *token* must identify a previously preinitialized environment that is dormant at the time of the call.

env_return_code (output)

A fullword integer which is set to the return code from the environment termination.

If the environment was initialized for a main routine or a subroutine, and the last CELQPIPI(*call_sub*) or CELQPIPI(*call_sub_addr*) issued stop semantics, the value of *env_return_code* is zero.

If the environment was initialized for a subroutine and the last CELQPIPI(*call_sub*) or CELQPIPI(*call_sub_addr*) did not terminate with stop semantics, *env_return_code* contains the same value as that in *sub_ret_code* from the last CELQPIPI(*call_sub*) or CELQPIPI(*call_sub_addr*).

Return Codes: Upon return, register 15 contains a return code indicating the success or failure of this request and can contain the following values:

- 0 The environment was activated and termination was requested.
- 4 Non-valid function code.
- 16 The *token* is not valid.
- 48 CELQPIPI was called from a non-64-bit environment.

Usage Notes:

- All resources obtained are released when the environment terminates.
- All routines loaded by Language Environment are deleted when the environment terminates.
- Subsequent references to *token* by preinitialization services result in an error indicating the token is not valid.

CELQPIPI(add_entry) — *add an entry to the PreInit table:* This invocation of *CELQPIPI* adds an entry for the environment represented by *token* in the Language Environment-maintained table. If a routine entry address is not provided, the routine name is used to dynamically load the routine and add it to the PreInit table. The PreInit table index for the new entry is returned to the calling routine.

Syntax

```
▶▶—CALL—CELQPIPI—(—add_entry—,—token—,—routine_name—,——————▶
▶—routine_entry—,—ceexptbl_index—)——————▶▶
```

add_entry (input)

A fullword function code (integer value = 6) containing the *add_entry* request.

token (input)

A doubleword with the value of the token associated with the environment that adds this new routine. This token is returned by a *CELQPIPI*(*init_main*), or *CELQPIPI*(*init_sub*) request.

The *token* must identify a previously preinitialized environment that is dormant at the time of the call.

routine_name (input)

A character string of length 8, left-justified and padded right with blanks, containing the name of the routine. To indicate the absence of the name, this field should be blank. If *routine_entry* is zero, this is used as the load name.

routine_entry (input/output)

The routine entry address that is added to the PreInit table. If *routine_entry* is zero on input, *routine_name* is used as the load name. On output, *routine_entry* is set to the load address of *routine_name*.

ceexptbl_index (output)

The doubleword index to the PreInit table where this routine was added. If the return code is nonzero, this value is indeterminate. The index starts at zero.

Return Codes: Upon return, register 15 contains a return code indicating the success or failure of this request and can contain one of the following values:

- 0 The routine was added to the PreInit table.
- 4 Non-valid function code.
- 12 The routine did not contain a valid Language Environment[®] entry prolog. Ensure that the routine was compiled with a current Language Environment enabled compiler. The PreInit table was not updated. The *routine_entry* is set to the address of the loaded routine.
- 16 The *token* is not valid.
- 20 The *routine_name* contains only blanks and the *routine_entry* was zero. The PreInit table was not updated.
- 24 The *routine_name* was not found or there was a load failure; the PreInit table was not updated.
- 28 The PreInit table is full. No routine was added to the table, nor was any routine loaded by Language Environment.
- 48 *CELQPIPI* was called from a non-64-bit environment.

CELQPIPI(delete_entry) — delete an entry from the PreInit table: This function deletes an entry from the PreInit table. The entry is then available for subsequent (add_entry) functions.

Syntax

```
▶▶CALLCELQPIPI(—delete_entry—,—token—,—ceexptbl_index—)▶▶
```

delete_entry (input)

fullword function code (integer value = 11) containing the delete_entry request

token (input)

a doubleword with the value of the token of the environment. This is the token returned by a CELQPIPI(init_main), or CELQPIPI(init_sub) request.

ceexptbl_index (input)

the index into the PreInit table of the entry to delete.

Return Codes: Upon return, R15 contains a return code indicating the success or failure of this request and may contain the following values:

- 0 The routine was deleted from the PreInit table
- 4 The function code is not valid.
- 16 The *token* is not valid
- 20 The PreInit table entry indicated by *ceexptbl_index* was empty.
- 24 The index passed is outside the range of the table.
- 28 The system request to delete the routine failed; the routine was not deleted from the PreInit table.
- 48 CELQPIPI was called from a non-64-bit environment.

Usage Notes:

- If the routine indicated by *ceexptbl_index* had been loaded by CELQPIPI, it will be deleted.

CELQPIPI(identify_entry) — identify an entry in the PreInit table: The invocation of this routine identifies the language of the entry point for a routine in the PreInit table.

Syntax

```
▶▶CALLCELQPIPI(—identify_entry—,—token—,—ceexptbl_index—,—▶▶
▶▶programming language—)▶▶
```

identify_entry (input)

A fullword containing the *identify_entry* function code (integer value=13).

token (input)

A doubleword with the value of the token of the environment. This is the token returned by a CELQPIPI(init_main) or CELQPIPI(init_sub) request.

ceexptbl_index (input)

A doubleword containing the index in the PreInit table of the entry to identify the programming language.

programming language **(output)**

A fullword with one of the following possible values:

- 3 C/C++
- 15 Language Environment-enabled assembler

Return Codes: Upon return, register 15 contains a return code indicating the success or failure of this request and can contain the following values:

- 0 The programming language has been returned.
- 4 Non-valid function code.
- 16 The *token* is not valid.
- 20 The PreInit table entry indicated by *ceexptbl_index* was empty.
- 24 The index passed is outside the range of the table.
- 48 CELQPIPI was called from a non-64-bit environment.

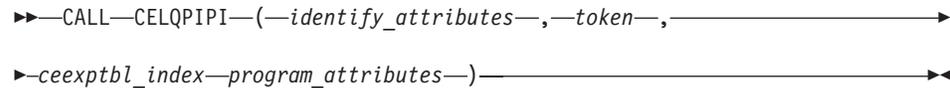
Usage Notes:

- The *programming_language* can be used by the driver program to determine the format of the parameter list specified on a *call_main*, for those cases where the language of the routine associated with *ceexptbl_index* is not known.
- When a PreInit table entry contains multiple languages, *programming_language* is the language of the entry point for the entry.

CELQPIPI(identify_attributes) — *identify the program attributes in the PreInit table:*

This invocation of CELQPIPI identifies the program attributes of a program in the PreInit table.

Syntax



identify_attributes **(input)**

A fullword function code (integer value = 16) containing the *identify_attributes* request

token **(input)**

A doubleword with the value of the *token* of the environment. This is the *token* returned by a *CELQPIPI(init_main)*, or *CELQPIPI(init_sub)*.

ceexptbl_index **(input)**

A doubleword containing the index in the PreInit table of the entry to identify the programming attributes.

program_attributes **(output)**

A fullword (32-bit) mask value will be returned indicating the following:

Table 26. program_attributes mask values

X'80000000'	The Preinitialization entry was loaded by Language Environment
X'40000000'	The Preinitialization entry could not be loaded, address not resolved
X'20000000'	The Preinitialization routine is not valid
X'10000000'	The function descriptor is not valid

Return Codes: Upon return, register 15 contains a return code indicating the success or failure of this request and can contain the following values:

- 0 The Preinitialization environment mask has been returned.
- 4 Non-valid function code.
- 16 The *token* is not valid.
- 20 The PreInit table entry indicated by *ceexptbl_index* was empty.
- 24 The index passed is outside the range of the table.
- 48 CELQPIPI was called from a non-64-bit environment.

Service Routines

Under Language Environment, you can specify service routines when executing a main routine or subroutine in the preinitialized environment. To use the routines, specify a list of addresses of the routines in a service routine vector as shown in Figure 37.

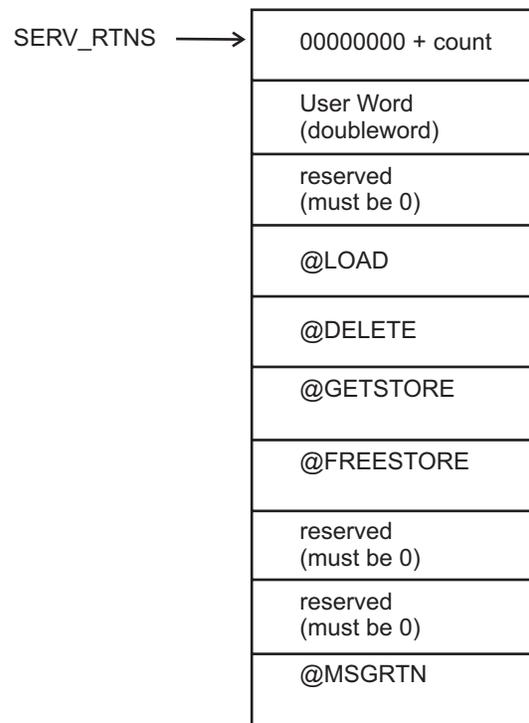


Figure 37. Service routines

The service routine vector is composed of a list of doubleword addresses of 64-bit XPLINK function pointers for routines that are used instead of Language Environment service routines. The list of addresses is preceded by the number of doubleword addresses in the list, as specified in the *count* field of the vector. The *service_rtms* parameter that you specify in calls to `CELQPIPI(init_main)` and `CELQPIPI(init_sub)` contains the address of the vector itself. If the pointer is specified as zero (0), Language Environment routines are used instead of the service routines shown in Figure 37.

The service routines must be 64-bit XPLINK functions. The function descriptors pointed to by the service vector are standard 64-bit function descriptors.

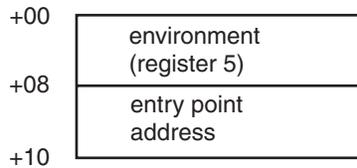


Figure 38. 64-bit function descriptors

When the service routine is called, Register 5 will contain the first 8 bytes in the function descriptor. The service routine will be called at the entry point address contained in the second 8 bytes of the function descriptor. Register 7 will be the return address to Language Environment. Register 4 will point to the caller's DSA, and the service routine can obtain a 64-bit XPLINK DSA using the usual methods.

In most cases, Register 4 points into the normal 64-bit XPLINK stack for the current TCB. In a few cases, Register 4 may point to some other fixed-size 64-bit stack, or to the regular 64-bit Language Environment stack for a different TCB. In these cases, there will be at least 4K of free space in the nonstandard stack to obtain a new XPLINK DSA.

The first three XPLINK parameters are passed in Registers 1, 2, and 3. Parameters 4 through 'n' are available in the caller's XPLINK argument area, starting 0x898 bytes past the input Register 4. The service routine can save the first three parameters (in registers 1, 2, and 3) in the caller's argument area starting at offset 0x880, if required.

These XPLINK function descriptions must remain accessible at the specified address from the time of the first CELQPIPI initialization call until the final CELQPIPI termination call has returned.

The @GETSTORE and @FREESTORE service routines must be specified together; if one is zero, the call to CELQPIPI(init_main) or CELQPIPI(init_sub) fails. The same is true for the @LOAD and @DELETE service routines. If you specify the @GETSTORE and @FREESTORE service routines, you do not have to specify the @LOAD and @DELETE service routines, and vice versa.

When replacing only the storage management routines without the program management routines, the user must be aware that they may not be accounting for all of the storage obtained on behalf of the application. Contents management obtains storage for the load module being loaded. This storage will not be managed by the user storage management routines.

00000000

A fullword zero at the start of the service routine vector.

Count A fullword binary number representing the number of doublewords that follow. The count does not include itself. In **Figure 37** the count is 9. If the count does include any of the reserved doublewords, those doublewords must be set to zero.

User Word

A doubleword that is passed to the service routines. The user word is provided as a means for your routine to communicate to the service routines.

@LOAD

This routine loads named routines for application management. The parameters that are passed contain the following:

Name_addr

The 64-bit address of the name of the module to load (input parameter).

User_word

A doubleword user field (input parameter).

Load_point

Either zero (0), or the 64-bit address where the @LOAD routine is to store the load point address of the loaded routine (input and output parameter). If not zero, it points to an 8-byte area. The 64-bit address of the load point is to be saved in this output area.

Entry_point

The 64-bit entry point address of the loaded routine (output parameter).

Module_size

The fixed binary(31) size of the module that was loaded (output parameter).

Return_code

The fullword return code from load (output).

Reason_code

The fullword reason code from load (output).

The return and reason codes are listed in the Table 27. AA should be the Abend (low 16 bits) code from the LOAD MACRO, and RR can be the reason code (low 16 bits) from the LOAD MACRO:

Table 27. Return and reason codes

Return Code	Reason Code	Description
0	0	Successful
0	3	Successful - loaded using SVC8
8	AARR	Unsuccessful - module not found
12	AARR	Unsuccessful - not enough storage
16	AARR	Unsuccessful
20	AARR	Unsuccessful - module came from unauthorized library (abend code 306 from LOAD)

@DELETE

This routine deletes routines for application management. The parameters that are passed contain the following:

Name_addr

The 64-bit address of the module name to be deleted (input parameter).

Name_length

A fixed binary(31) length of module name (input parameter).

User_word

A doubleword user field (input parameter).

Rsvd_word

A fullword reserved for future use (*any value that might be present in this parameter should not be relied upon*).

Return_code

The fullword return code from the delete service (output parameter).

Reason_code

The fullword reason code from the delete service (output parameter).

The return and reason codes are listed in Table 28.

Table 28. Return and reason codes.

Return Code	Reason Code	Description
0	0	Successful
4	4	Unsuccessful - DELETE failed

@GETSTORE

This routine allocates storage on behalf of the Language Environment storage manager. Three types of storage can be requested: below the 16MB line, below the 2GB bar, and above the 2GB bar. The parameter list that is passed contains the following:

PISA_Addr

A doubleword address of the PreInit storage attributes (PISA) control block (input parameter). This control block, defined in macro CELQPIDF, contains the attributes that the GETSTORE routine is to use when obtaining the storage. Table 29 shows the control block field descriptions.

Table 29. Preinit Storage Attributes Control Block Field Descriptions

Name	Description
PisaVersion	A fullword field that contains the version number of this PISA. The only valid value for this field is currently 1, but other values may be added in the future when new storage attributes are added to this interface. The GETSTORE service routine should check this field to verify that it supports the specified version; if it cannot support the version, it should return with a return code of 8.
PisaAmount	An unsigned doubleword that contains the amount of storage requested. For below-the-bar storage, the amount of storage to obtain is in the number of bytes. For above-the-bar storage, the size of the memory object to obtain is in the number of megabytes. Note: Above-the-bar storage returned by @GETSTORE must be aligned on a 1MB boundary.

Table 29. Preinit Storage Attributes Control Block Field Descriptions (continued)

Name	Description
PisaFlags	<p>A doubleword flag area. The flags are defined as follows:</p> <p>PisaBelowtheLine Bit zero in PisaFlags is ON if the requested storage is required to be below the 16MB line.</p> <p>PisaBelowtheBar Bit one in PisaFlags is ON if the requested storage is required to be below the 2GB bar.</p> <p>PisaAbovetheBar Bit two in PisaFlags is ON if the requested storage is required to be above the 2GB bar.</p> <p>PisaGuardLoc Bit three in PisaFlags specifies whether the guard location is at the low virtual end or the high virtual end of the memory object. Bit three is OFF if the guard areas are created starting from the origin of the memory object; that is, from the low virtual end. Bit three is ON if the guard areas are created at the end of the memory object, that is, at the high virtual end. This flag is used for above-the-bar storage only, and only has meaning when PisaGuardSize is non-zero. Failure to guard the memory object as requested may cause Language Environment's stack management features to work incorrectly.</p> <p>PisaPageFrameSize1MEG Bit four in PisaFlags is ON if the memory object is to be backed by 1 megabyte page frames (equivalent to IARV64 PageFrameSize=1MEG). When both bits four and five are off, the memory object will be backed by 4KB page frames (equivalent to IARV64 PageFrameSize=4K). This flag is used for above-the-bar storage only. Failure to provide storage with this attribute may have a performance impact in your application.</p> <p>PisaPageFrameSizeMAX Bit five in PisaFlags is ON if the memory object is to be backed by the largest page frame size supported (equivalent to IARV64 PageFrameSize=MAX). When both bits four and five are off, the memory object will be backed by 4KB page frames (equivalent to IARV64 PageFrameSize=4K). This flag is used for above-the-bar storage only. Failure to provide storage with this attribute may have a performance impact in your application.</p>
PisaSubpool_no	<p>A fixed binary(31) subpool number from 0 to 127. Language Environment allocates storage from the process-level storage pools. This field is used for below-the-bar storage only.</p>
PisaDumpPriority	<p>A fullword dump priority for memory objects. This field is used for above-the-bar storage only. Failure to provide storage with the requested dump priority may result in a dump that does not contain useful diagnostic information.</p>
PisaGuardSize	<p>A doubleword indicating the number of megabytes of guard area to be created at the high or low end of the memory object. This field is used for above-the-bar storage only. Failure to guard the memory object as requested may cause Language Environment's stack management features to work incorrectly.</p>
PisaUserTKN	<p>A doubleword token to be associated with a group of memory objects. This can be used on a later FREESTORE request to free all memory objects associated with this value. This field is used for above-the-bar storage only.</p> <p>Note: Non-authorized callers will not be able to directly use the UserTKN provided by Language Environment because Language Environment uses the authorized word in the high half. Refer to the sample @GETSTORE routine to see one way of handling this.</p>

User_word

A doubleword field containing the user word supplied in the Service Routine Vector (input parameter).

Stg_address

A doubleword address of the storage obtained or zero (output parameter) .

Obtained

An unsigned doubleword that contains the number of bytes (below the bar) or megabytes (above the bar) obtained (output parameter).

Note: Storage must be obtained in a key that is compatible with the application.

Return code

The fullword return code from the @GETSTORE service (output parameter).

Reason code

The fullword reason code from the @GETSTORE service (output parameter).

The return and reason codes are listed in Table 30.

Table 30. Return and reason codes.

Return Code	Reason Code	Description
0	0	Successful
8	0	The specified version of the PISA is not supported by this service routine.
16	0	Unsuccessful - uncorrectable error occurred.

@FREESTORE

This routine frees storage on behalf of the Language Environment storage manager. The parameter list that is passed contains the following:

Stg_address

The doubleword address of the storage or memory object to free (input parameter) .

Note: Stg_address will be zero when MatchUserTKN is ON.

Amount

An unsigned doubleword that contains the amount of storage in bytes to free (input parameter). This field is used for below-the-bar storage only.

Subpool_no

The fixed binary(31) subpool number from 0 to 127 (input parameter). This parameter is used for below-the-bar storage only.

Flags A doubleword flag area (input parameter), defined as follows:

MatchUserTKN

Bit zero in Flags indicates how the value specified in parameter UserTKN is to be used. If MatchUserTKN is ON, the @FREESTORE routine is expected to free all memory objects that are associated with this user token (equivalent to IARV64

MATCH=USERTKN). If MatchUsertkn is OFF, @FREESTORE is to use this user token when freeing a single memory object with the origin in Stg_address (equivalent to MATCH=SINGLE). This flag is used for above-the-bar storage only.

UserTkn

A doubleword token that identifies the memory object or a group of memory objects to be detached (input parameter). This parameter is used for above-the-bar storage only.

User word

A doubleword field containing the user word supplied in the Service Routine Vector (input parameter).

Return code

The fullword return code from the @FREESTORE service (output parameter) .

Reason code

The fullword reason code from the @FREESTORE service (output parameter).

The return and reason codes are listed in Table 31.

Table 31. Return and reason codes.

Return Code	Reason Code	Description
0	0	Successful
16	0	Unsuccessful - uncorrectable error occurred.

@MSGRTN

This routine allows error messages to be processed by the caller of the application.

If the message pointer is zero, your message routine is expected to return the size of the line to which messages are written (in the line_length field). This allows messages to be formatted correctly; that is, broken at places such as blanks.

Message

A pointer to the first byte of text that is printed, or zero (input parameter).

Msg_len

The fixed binary(31) length of the message (input parameter).

User word

A doubleword user field (input parameter).

Line_length

The fixed binary(31) size of the output line length (output parameter). This is used when Message is zero.

Return code

The fullword return code from the @MSGRTN service (output parameter).

Reason code

The fullword reason code from the @MSGRTN service (output parameter).

The return and reason codes are listed in Table 32.

Table 32. Return and reason codes.

Return Code	Reason Code	Description
0	0	Successful
16	4	Unsuccessful - uncorrectable error occurred.

An example program invocation of CELQPIPI

Figure 39 contains a sample CELQPIPI program invocation with an AMODE 64 PreInit assembler driver program.

The assembler driver, CEEWQPIP, invokes CELQPIPI to:

- Initialize a main routine environment under Language Environment, with service routines CEEWQLOD, CEEWQDEL, CEEWQGST, CEEWQFST, and CEEWQMSG replacing any Language Environment LOAD, DELETE, GETSTORE, FREESTORE, and MSGRTN service routines, respectively.
- Load and call CEEWQPMA, a reentrant HLL main routine written in C, which causes the replacement service routines to be run.
- Terminate the Language Environment environment.

```

*****
*
*
* =====  -----
* CEEWQPIP - AMODE64 PreInit Driver with service routines
* =====  -----
*
*
* This sample PreInit driver illustrates the use of LOAD, DELETE
* GETSTORE, FREESTORE, and MSGRTN replacement routines in an
* AMODE64 driver.
*
*
* This sample driver does the following:
*
* 1) CELQPIPI INIT_MAIN request, with a single-entry PreInit table
*    containing "CEEWQPMA", which is a C main() program.
*
*    Service Routine Vector contains:
*
*      - Count = 9
*      - User Word points to a below-the-bar work area for
*        the LOAD and DELETE replacement routines
*      - LOAD replacement routine is "CEEWQLOD"
*      - DELETE replacement routine is "CEEWQDEL"
*      - GETSTORE replacement routine is "CEEWQGST"
*      - FREESTORE replacement routine is "CEEWQFST"
*      - MSGRTN replacement routine is "CEEWQMSG"
*      - All other doublewords in the service vector are 0
*
*
*

```

Figure 39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment (Part 1 of 9)

```

* 2) CELQPIPI CALL_MAIN request, for the only row (0) in the
*   PreInit table.
*
*   - The Runtime options are "POSIX(ON)"
*
*   - The parms passed to the C main() program are:
*
*       argc    = 3
*       argv[0] = "ceewqpma"
*       argv[1] = "Parm1"
*       argv[2] = "Parm2"
*
*
* 3) CELQPIPI TERM request
*
*
* Note: This text deck must be bound with the following:
*
*   CEEWQLOD -- LOAD replacement routine
*   CEEWQDEL -- DELETE replacement routine
*   CEEWQGST -- GETSTORE replacement routine
*   CEEWQFST -- FREESTORE replacement routine
*   CEEWQMSG -- MSGRTN replacement routine
*   CELQSTRT (from SCEEBND2)
*   CELQETBL (from SCEEBND2)
*   CELQLLST (from SCEEBND2)
*
* Note: At runtime, the C main() program (CEEWQPMA) must be available
*       for LOAD.
*
*
*****

```

Figure 39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment (Part 2 of 9)

```

*
CEEWQPIP CSECT      ,
CEEWQPIP AMODE     64
CEEWQPIP RMODE     31
          SYSSTATE AMODE64=YES
          SAM64     ,
*
*
*   Standard 64-bit entry linkage
*   -----
*
STMG     R14,R12,SAVF4SAG64RS14-SAVF4SA(R13)  Save caller regs
BASR     R11,0                                Set up basereg
USING    *,R11                                Addressability
GETMAIN  RU,LV=DSA_L                          Obtain DSA
STG      R13,SAVF4SAPREV-SAVF4SA(,R1)        Set backchain
STG      R1,SAVF4SANEXT-SAVF4SA(,R13)        Set fwd chain
MVC      SAVF4SAID-SAVF4SA(R4,R13),=A(SAVF4SAID_VALUE) "F4SA"
LGR      R13,R1                                Set up DSAREg
USING    DSA,R13                              Addressability
*
*
*   Issue LOAD for CELQPIPI (will ABEND if LOAD fails)
*   -----
*
WTO      'CEEWQPIP: LOADING CELQPIPI',ROUTCDE=11
*
LOAD     EP=CELQPIPI                          LOAD LE main module
NG       R0,=X'00000000FFFFFFFF'             Clear low (AMODE64) bit
STG      R0,CELQPIPI_EP                      Save CELQPIPI E.P. Address
*
*
*   Set up Service Routine Vector and parm
*   -----
*
MVC      SV_DYNAMIC,SV_STATIC                Copy over into DSA
LA       R15,USER_AREA                       Point to 1000-byte user area
STG      R15,SV_UWORD                        SV user word -> user area
LA       R15,SV_DYNAMIC                      Address of modifiable SV
STG      R15,SERVICE_RTNS                   Save as parm for INIT_MAIN
*
*
*   Do CELQPIPI INIT_MAIN
*   -----
*
WTO      'CEEWQPIP: Doing CELQPIPI INIT_MAIN',ROUTCDE=11
*
LG       R15,CELQPIPI_EP                     Address of CELQPIPI E.P.
*
CALL     (15),                                X
          (INIT_MAIN,                        CELQPIPI INIT_MAIN request X
          CEEXPTBL_ADDR,                    Address of CELQPIPI table X
          SERVICE_RTNS,                    Address of service rtn vector X
          TOKEN),                            Token from INIT_MAIN X
          MF=(E,CALL_PL)

```

Figure 39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment (Part 3 of 9)

```

*
*
*   Check results of INIT_MAIN
*
LTGR R2,R15          Check CELQPIPI R/C
BZ   INIT_OK        Go do CALL_MAIN, if OK
WTO  'CEEWQPIP: CELQPIPI INIT_MAIN failed',ROUTCDE=11
MVC  RC_MSGD,RC_MSGC Create modifiable message text
LH   R15,RC_MSGCN(R2) Get printable R/C
STH  R15,RC_MSGDN   Save in modifiable text
MVC  WTO_PLD(WTO_PLL),WTO_PLC Create modifiable WTO plist
WTO  TEXT=RC_MSGD,ROUTCDE=11,MF=(E,WTO_PLD)
B    DO_TERM        Bypass CALL_MAIN
INIT_OK EQU *
*
*
*   Do CELQPIPI CALL_MAIN
*   -----
*
WTO  'CEEWQPIP: Doing CELQPIPI CALL_MAIN',ROUTCDE=11
*
LG   R15,CELQPIPI_EP Address of CELQPIPI E.P.
*
CALL (15),
      (CALL_MAIN,          CELQPIPI CALL_MAIN request      X
      CEEXPTL_INDEX,      CELQPIPI table index (= 0)      X
      TOKEN,              Token from INIT_MAIN          X
      RUNTIME_OPTS,       Runtime Options                  X
      PARM_PTR,           Ptr to C main() parmlist         X
      ENCLAVE_RETURN_CODE, Enclave return code            X
      ENCLAVE_REASON_CODE, Enclave reason code            X
      APPL_FEEDBACK_CODE), Application feedback code      X
      MF=(E,CALL_PL)
*
*
*   Check results of CALL_MAIN
*
LTR  R2,R15          Check CELQPIPI R/C
BZ   CALL_OK        Bypass message, if OK
WTO  'CEEWQPIP: CELQPIPI CALL_MAIN failed',ROUTCDE=11
MVC  RC_MSGD,RC_MSGC Create modifiable message text
LH   R15,RC_MSGCN(R2) Get printable R/C
STH  R15,RC_MSGDN   Save in modifiable text
MVC  WTO_PLD(WTO_PLL),WTO_PLC Create modifiable WTO plist
WTO  TEXT=RC_MSGD,ROUTCDE=11,MF=(E,WTO_PLD)
CALL_OK EQU *
*
*

```

Figure 39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment (Part 4 of 9)

```

*      Do CELQPIPI TERM
*      -----
*
DO_TERM EQU *
WTO     'CEEWQPIP: Doing CELQPIPI TERM',ROUTCDE=11
*
LG      R15,CELQPIPI_EP           Address of CELQPIPI E.P.
*
CALL    (15),                      X
        (TERM,                      CELQPIPI TERM request      X
         TOKEN,                      Token from INIT_MAIN      X
         ENV_RETURN_CODE),          Environment return code    X
         MF=(E,CALL_PL)
*
LTR     R2,R15                     Check CELQPIPI R/C
BZ      TERM_OK                    Bypass message, if OK
WTO     'CEEWQPIP: CELQPIPI TERM failed',ROUTCDE=11
TERM_OK EQU *
*
*      Return to caller with R/C=0
*      -----
*
WTO     'CEEWQPIP: Returning',ROUTCDE=11
*
LGR     R1,R13                     Addr for FREEMAIN
LG      R13,SAVF4SAPREV-SAVF4SA(,R13) Caller's DSA addr
FREEMAIN RU,A=(1),LV=1024          Get rid of DSA
LA      R15,0                      Set R/C = 0
LG      R14,SAVF4SAG64RS14-SAVF4SA(,R13) Restore R14
LMG     R0,R12,SAVF4SAG64RS0-SAVF4SA(R13) Restore R0-R12
SAM31   ,
BR      R14                        Return to caller
*
*
*      -----
*      Static constants
*      -----
*
LTORG   ,
*
INIT_MAIN DC F'1'                   Initialize for main routines
CALL_MAIN DC F'2'                   Call main routine
TERM      DC F'5'                   Terminate
*
RC_MSGC   DS 0CL28
          DC AL2(26)
          DC C'          CELQPIPI R/C: nn'
RC_MSGCEN DC C'00..04..08..12..16..20..24..28..32..36..'
          DC C'40..44..48..52..56..60..64..68..72..76..'
*
WTO_PLC   WTO TEXT=RC_MSGD,ROUTCDE=11,MF=L
*
CEEXPTBL_ADDR DC AD(CEEXPTBL)      Address of PIPi table
CEEXPTL_INDEX DC AD(0)             1st row of CEEXPTBL = 0
*
RUNTIME_OPTS DC CL255'POSIX(ON)'
PARM_PTR    DC AD(P_PLIST)
*
*

```

Figure 39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment (Part 5 of 9)

```

*          CELQPIPI service routine vector (static copy)
*
SV_STATIC  DS      0D
SV_RES     DC      0XL80      10 doublewords
SV_COUNT   DC      A(0)      Reserved (must be 0)
SV_COUNT   DC      A(9)      9 entries in service vector
SV_UWORD_  DC      AD(*-*)   Will be filled in at runtime
SV_2       DC      AD(0)     Reserved (must be 0)
SV_LOAD    DC      AD(FD_L)   Pointer to FD for LOAD routine
SV_DELETE  DC      AD(FD_D)   Pointer to FD for DELETE routine
SV_GETSTOR DC      AD(FD_G)   Pointer to FD for GETSTORE routine
SV_FREESTOR DC      AD(FD_F)   Pointer to FD for FREESTOR routine
SV_7       DC      AD(0)     Reserved (must be 0)
SV_8       DC      AD(0)     Reserved (must be 0)
SV_MSGRTN  DC      AD(FD_M)   Pointer to FD for MSGRTN routine
*
*          Function Descriptor for CEEWQLOD (load replacement)
*
EXTRN      CEEWQLOD
CEEWQLOD   XATTR   LINKAGE(XPLINK),REFERENCE(CODE)
*
FD_L       DS      0D
FD_L       DS      0CL16
FD_L       DC      RD(CEEWQLOD) PSECT Address
FD_L       DC      AD(CEEWQLOD) E.P. Address
*
*

```

Figure 39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment (Part 6 of 9)

```

*           Function Descriptor for CEEWQDEL (delete replacement)
*
CEEWQDEL   EXTRN   CEEWQDEL
           XATTR   LINKAGE(XPLINK),REFERENCE(CODE)
*
FD_D       DS      0D
           DS      0CL16
           DC      RD(CEEWQDEL)  PSECT address
           DC      AD(CEEWQDEL)  E.P. Address
*
*           Function Descriptor for CEEWQGST (getstore replacement)
*
CEEWQGST   EXTRN   CEEWQGST
           XATTR   LINKAGE(XPLINK),REFERENCE(CODE)
*
FD_G       DS      0D
           DS      0CL16
           DC      RD(CEEWQGST)  PSECT address
           DC      AD(CEEWQGST)  E.P. Address
*
*           Function Descriptor for CEEWQFST (freestore replacement)
*
CEEWQFST   EXTRN   CEEWQFST
           XATTR   LINKAGE(XPLINK),REFERENCE(CODE)
*
FD_F       DS      0D
           DS      0CL16
           DC      RD(CEEWQFST)  PSECT address
           DC      AD(CEEWQFST)  E.P. Address
*
*           Function Descriptor for CEEWQMSG (msggrtn replacement)
*
CEEWQMSG   EXTRN   CEEWQMSG
           XATTR   LINKAGE(XPLINK),REFERENCE(CODE)
*
FD_M       DS      0D
           DS      0CL16
           DC      RD(CEEWQMSG)  PSECT address
           DC      AD(CEEWQMSG)  E.P. Address
*
*

```

Figure 39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment (Part 7 of 9)

```

*          CELQPIPI table (with 1 static entry)
*
CEEXPTBL  CELQPIT  ,          Start of CELQPIPI table
          CELQPITY CEEWQPMA,0  Dynamically load CEEEQPMA
          CELQPITS ,          End of CELQPIPI table

*
*
*          Parmlist for C main()
*
          DS      0D
P_PLIST   DS      0XL16
P_ARGC    DC      FD'3'          argc = 3
P_ARGV    DC      AD(P_ARG)     pointer to argv
*
          DS      0D
P_ARG     DS      0XL16          argv
P_ARG0    DC      AD(P_NAMEBUF) argv[0]
P_ARG1    DC      AD(P_ARGBUF1) argv[1]
P_ARG2    DC      AD(P_ARGBUF2) argv[2]
          DC      AD(0)

*
P_NAMEBUF DS      0CL9          argv[0]
          DC      CL8'ceewqpm' program name
          DC      XL1'0'         NULL terminator

*
P_ARGBUF1 DS      0CL6          argv[1]
          DC      CL5'Parm1'    1st parm = "Parm1"
          DC      XL1'0'         NULL terminator

*
P_ARGBUF2 DS      0CL6          argv[2]
          DC      CL5'Parm2'    2nd parm = "Parm2"
          DC      XL1'0'         NULL terminator
          DS      0D

*
*

```

Figure 39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment (Part 8 of 9)

```

*
* -----
* DSA (below the bar, due to GETMAIN)
* -----
*
DSA          DSECT ,
             DS    CL(SAVF4SA_LEN) F4 savearea
             DS    0D
*
CELQPIPI_EP  DS    AD           Address of CELQPIPI E.P.
TOKEN        DS    AD           Token (from INIT_MAIN)
SERVICE_RTNS DS    AD           Address of Service vector
APPL_FEEDBACK_CODE DS 2AD       Fdbk code from CALL_MAIN
ENCLAVE_RETURN_CODE DS  F       Return code from CALL_MAIN
ENCLAVE_REASON_CODE DS  F       Reason code from CALL_MAIN
ENV_RETURN_CODE DS  F         Rtn code from CELQPIPI TERM
*
RC_MSGD      DS    0CL28        Return code message for WTO
             DS    CL2          Text length = 26
             DS    CL24         '          CELQPIPI R/C: '
RC_MSGDN     DS    CL2          'nn'
*
             DS    0D
SV_DYNAMIC   DS    0XL(L'SV_STATIC) Modifiable service vector
             DS    AD           0000000 + count
SV_UWORD     DS    AD           User word
             DS    AD           doubleword 2
             DS    AD           doubleword 3
             DS    AD           doubleword 4
             DS    AD           doubleword 5
             DS    AD           doubleword 6
             DS    AD           doubleword 6
             DS    AD           doubleword 7
             DS    AD           doubleword 8
             DS    AD           doubleword 9
*
             DS    0D
CALL_PL      CALL  ,(,,,,,,),MF=L 8-parm CALL parmlist
*
             DS    0D
WTO_PLD     WTO   TEXT=RC_MSGD,ROUTCDE=11,MF=L
WTO_PLL     EQU   *-WTO_PLD
*
             DS    0D
USER_AREA   DS    XL1000        1000-byte below-the-bar
*                                     workarea for LOAD and
*                                     DELETE replacement routines
*
             DS    0D
DSA_L       EQU   *-DSA         Length of DSA
*
* -----
* Control Blocks and equates
* -----
*
YREGS      ,
IHASAVER  ,
*
END

```

Figure 39. AMODE 64 Assembler Driver That Creates a Preinitialized Environment (Part 9 of 9)

A sample AMODE 64 Assembler LOAD replacement service routine, CEEWQLOD, can be found in SCEESAMP(CEEWQLOD).

A sample AMODE 64 Assembler DELETE replacement service routine, CEEWQDEL, can be found in SCEESAMP(CEEWQDEL).

A sample AMODE 64 Assembler GETSTORE replacement service routine, CEEWQGST, can be found in SCEESAMP(CEEWQGST).

A sample AMODE 64 Assembler FREESTORE replacement service routine, CEEWQFST, can be found in SCEESAMP(CEEWQFST).

A sample AMODE 64 Assembler MSGRTN replacement service routine, CEEWQMSG, can be found in SCEESAMP(CEEWQMSG).

Figure 40 contains a sample AMODE 64 C main routine.

```

/*****
*
*
* =====  -----
* CEEWQPMA -- Sample AMODE64 PreInit program
* =====  -----
*
*
* This program is invoked with two parms (always assumed present)
* by the CEEWQPIP AMODE64 PreInit driver program.
*
*
* Processing is:
*
* 1) Print out POSIX(ON/OFF) state to verify POSIX(ON) runtime
*    option.
*
* 2) Print out first two parms
*
* 3) Issue perror() to cause a LOAD request (and later DELETE)
*
*
*****/

#define _XOPEN_SOURCE_EXTENDED 1

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
    printf("\nCEEWQPMA: Called -- __isPosixOn()=%d\n", __isPosixOn());

    printf("CEEWQPMA: main() argc    = %d\n",    argc );
    printf("                argv[0]= \"%s\"\n",    argv[0];
    printf("                argv[1]= \"%s\"\n",    argv[1];
    printf("                argv[2]= \"%s\"\n\n",    argv[2];

    errno = ESTALE;
    perror("CEEWQPMA: Test message");

    printf("\nCEEWQPMA: Returning\n");

    return 0;
}

```

Figure 40. AMODE 64 C Main PreInit Routine

Part 5. Appendixes

Appendix. Accessibility

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

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

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer or Library Server versions of z/OS books in the Internet library at:

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

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

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always

present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

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

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

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

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

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

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

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

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

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

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
 2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
 3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.
- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Permission Notice

This book includes information about certain callable service stub and linkage-assist (stub) routines contained in specific data sets that are intended to be bound or link-edited with code and run on z/OS systems. In connection with your authorized use of z/OS, you may bind or link-edit these stubs into your modules and distribute your modules with the included stubs for the purposes of developing, using, marketing and distributing programs conforming to the documented programming interfaces for z/OS, provided that each stub is included in its entirety, including any IBM copyright statements. These stubs have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply the reliability, serviceability, or function of these stub programs. The stubs referred to in this book are contained in one or more of the following data sets:

- CEE.SAFHFORT
- CEE.SCEEBIND
- CEE.SCEEEND2
- CEE.SCEECP
- CEE.SCEELKED
- CEE.SCEELKEX
- CEE.SCEEOBJ
- CEE.SCEESPC
- CEE.SIBMAM24
- CEE.SIBMCALL
- CEE.SIBMCAL2
- CEE.SIBMMATH
- CEE.SIBMTASK

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of Language Environment in z/OS.

Trademarks

IBM, the IBM logo, and `ibm.com` are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

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

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

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

Bibliography

This section lists the books in the Language Environment library and other publications that may be helpful when using Language Environment.

Language Products Publications

z/OS Language Environment

- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Run-Time Application Migration Guide*, GA22-7565
- *z/OS Language Environment Writing Interlanguage Communication Applications*, SA22-7563
- *z/OS Language Environment Run-Time Messages*, SA22-7566
- *z/OS Language Environment Vendor Interfaces*, SA22-7568
- *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*, SA22-7569

z/OS XL C/C++

- *z/OS XL C/C++ Language Reference*, SC09-4815
- *z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer*, GC09-4913
- *z/OS XL C/C++ Programming Guide*, SC09-4765
- *z/OS XL C/C++ User's Guide*, SC09-4767
- *z/OS XL C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS XL C/C++ Messages*, GC09-4819
- *Standard C++ Library Reference*, SC09-4949

z/OS Metal C Runtime Library

- *z/OS Metal C Programming Guide and Reference*, SA23-2225

Enterprise COBOL for z/OS, V4R2

- *Enterprise COBOL for z/OS, V4R2, Licensed Program Specifications*, GI11-7871
- *Enterprise COBOL for z/OS, V4R2, Customization*, SC23-8526
- *Enterprise COBOL for z/OS, V4R2, Language Reference*, SC23-8528
- *Enterprise COBOL for z/OS, V4R2, Programming Guide*, SC23-8529
- *Enterprise COBOL for z/OS, V4R2, Compiler and Runtime Migration Guide*, GC23-8527

Enterprise COBOL for z/OS, V3R4

- *Enterprise COBOL for z/OS, V3R4, Licensed Program Specifications*, GC27-1411
- *Enterprise COBOL for z/OS, V3R4, Customization*, GC27-1410
- *Enterprise COBOL for z/OS, V3R4, Language Reference*, SC27-1408
- *Enterprise COBOL for z/OS, V3R4, Programming Guide*, SC27-1412
- *Enterprise COBOL for z/OS, V3R4, Compiler and Runtime Migration Guide*, GC27-1409

COBOL for OS/390 & VM

- *COBOL for OS/390 & VM Licensed Program Specifications*, GC26-9044
- *COBOL for OS/390 & VM Customization under OS/390*, GC26-9045

- *COBOL for OS/390 & VM Language Reference*, SC26-9046
- *COBOL for OS/390 & VM Programming Guide*, SC26-9049

Debug Tool

- Debug Tool documentation is available at: <http://www.ibm.com/software/awdtools/debugtool/>

VS FORTRAN Version 2

- *Language Environment for MVS & VM Fortran Run-Time Migration Guide*, SC26-8499
- *Language and Library Reference*, SC26-4221
- *Programming Guide for CMS and MVS*, SC26-4222

Enterprise PL/I for z/OS

- *Enterprise PL/I for z/OS, V3R9, Licensed Program Specifications*, GC27-1455
- *Enterprise PL/I for z/OS, V3R9, Programming Guide*, SC27-1457
- *Enterprise PL/I for z/OS, V3R9, Language Reference*, SC27-1460
- *Enterprise PL/I for z/OS, V3R9, Migration Guide*, GC27-1458
- *Enterprise PL/I for z/OS, V3R9, Messages and Codes*, SC27-1461

PL/I for MVS & VM

- *PL/I for MVS & VM Licensed Program Specifications*, GC26-3116
- *PL/I for MVS & VM Programming Guide*, SC26-3113
- *PL/I for MVS & VM Language Reference*, SC26-3114
- *PL/I for MVS & VM Reference Summary*, SX26-3821
- *PL/I for MVS & VM Compiler and Run-Time Migration Guide*, SC26-3118
- *PL/I for MVS & VM Installation and Customization under MVS*, SC26-3119
- *PL/I for MVS & VM Compile-Time Messages and Codes*, SC26-3229
- *PL/I for MVS & VM Diagnosis Guide*, SC26-3149

High Level Assembler for MVS & VM & VSE

- *Programmer's Guide, MVS & VM Edition*, SC26-4941

Related Publications

CICS

- *CICS Transaction Server for z/OS Installation Guide*, GC34-6224
- *CICS Operations and Utilities Guide*, SC34-6229
- *CICS Problem Determination Guide*, SC34-6239
- *CICS Resource Definition Guide*, SC34-6228
- *CICS Application Programming Guide*, SC34-6231
- *CICS Application Programming Reference*, SC34-6232
- *CICS System Definition Guide*, SC34-6226

DB2

- *Database 2 Application Programming and SQL Guide*, SC26-4377

DFSMS/MVS

- *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643
- *z/OS DFSMS DFM Guide and Reference*, SC26-7395
-

IPCS

- *z/OS MVS IPCS User's Guide*, SA22-7596
- *z/OS MVS IPCS Commands*, SA22-7594
- *z/OS MVS IPCS Customization*, SA22-7595

DFSORT

- *z/OS DFSORT Application Programming Guide, SC26-7523*

IMS/ESA

- *IMS Version 8: Application Programming: Design Guide, SC27-1287*
- *IMS Version 8: Application Programming: Database Manager, SC27-1286*
- *IMS Version 8: Application Programming: Transaction Manager, SC27-1289*
- *IMS Version 8: Application Programming: EXEC DLI Commands for CICS and IMS Version 8, SC27-1288*

z/OS

- *z/OS Introduction and Release Guide, GA22-7502*
- *z/OS Program Directory, GI10-0670*
- *z/OS Planning for Installation, GA22-7504*
- *z/OS Information Roadmap, SA22-7500*
- *z/OS Hot Topics Newsletter, GA22-7501*
- *z/OS Licensed Program Specifications, GA22-7503*

z/OS ISPF

- *z/OS ISPF Dialog Tag Language Guide and Reference, SC34-4824*
- *z/OS ISPF Planning and Customizing, GC34-4814*
- *z/OS ISPF Dialog Developer's Guide and Reference, SC34-4821*

z/OS UNIX System Services

- *z/OS UNIX System Services User's Guide, SA22-7801*
- *z/OS UNIX System Services Command Reference, SA22-7802*
- *z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA22-7803*
- *z/OS UNIX System Services Planning, GA22-7800*

z/OS TSO/E

- *z/OS TSO/E Customization, SA22-7783*
- *z/OS TSO/E Programming Services, SA22-7789*
- *z/OS TSO/E System Programming Command Reference, SA22-7793*

Index

Special characters

- #pragma directives
 - See pragma
- c89 utility
 - bind object modules 41
 - interface to the binder 41

A

- abnormal termination
 - See abends
- above-the-line
 - See 16M line
- accessibility 199
- addressing mode
 - See AMODE
- application
 - See also enclave
 - binding using c89 41
 - invoking MVS executable programs from a z/OS UNIX shell 43
 - placing MVS load modules in the HFS 43
 - running
 - from the z/OS UNIX shell 43
 - under batch 44
 - under MVS batch 44
- arithmetic
 - See math services
- automatic library call processing
 - See library call processing

B

- below-the-line
 - See 16M line
- binder
 - module name 37
- binder interface
 - c89 utility 41
- bit manipulation routines 145
- BPXBATCH program
 - invoking from TSO/E 44
 - running an executable HFS file under batch 44

C

- C
 - #pragmas
 - See pragma
 - functions
 - See main routine
 - See subroutine
 - L-names
 - See L-names
 - pragma
 - See pragma

- C (*continued*)
 - S-names
 - See S-names
 - system programming facilities
 - See system programming facility, C
- c89 utility
 - See z/OS UNIX, c89 utility
- CAA (common anchor area)
 - See common anchor area (CAA)
- CALL statement
 - for PL/I
 - See FETCH statement
- call_main
 - See preinitialization facility, CELQPIPI(call_main)
- call_sub
 - See preinitialization facility, CELQPIPI(call_sub)
- call_sub_addr
 - See preinitialization facility, CELQPIPI(call_sub_addr)
- CEEBLDTX utility
 - error messages 128
- CEEBXITA assembler user exit
 - interface to
 - See CXIT control block
- CEECXITA assembler user exit
 - See CEEBXITA assembler user exit
- CELQPIPI
 - See preinitialization facility
- CMOD CLIST
 - See LINK command for TSO
- command
 - syntax diagrams xii
- common environment, introduction 4
- condition handling
 - See also stack, frame
 - condition step of
 - See condition step
 - termination imminent step of
 - See termination imminent step
- CPPL (command processor parameter list)
 - See command processor parameter list (CPPL)

D

- definition side-deck 26
- disability 199
- DLLs (Dynamic Link Libraries) 13
 - application 13
 - applications 13
 - binding a DLL 26
 - binding a DLL application 26
 - C or C++ example 16
 - calling explicitly 15
 - calling implicitly 14
 - creating 22
 - #pragma export 23
 - C 22

- DLLs (Dynamic Link Libraries) (*continued*)
 - creating (*continued*)
 - description 22
 - exporting functions 23
 - entry point 30
 - example 29
 - freeing 22
 - function 13
 - load-on-call 14
 - loading 20
 - managing the use of 20
 - performance 31
 - restrictions 30
 - sharing among application executable files 21
 - using 27
 - variable 13
- DSA (dynamic save area)
 - See stack, frame
- dynamic save area (DSA)
 - See DSA (dynamic save area)

E

- end_seq
 - See preinitialization facility, CELQPIPI(end_seq)
- environment, common 4
- executable files
 - invoking MVS executable programs from a z/OS UNIX shell 43
 - placing MVS load modules in the HFS 43
 - running
 - from the z/OS UNIX shell 43
 - under batch 44
 - under MVS batch 44
 - exporting functions 14

F

- functions 14
 - exported 14
 - imported 14

G

- global error table
 - See condition handling

H

- heap storage
 - See also additional heap
 - performance improvement, using
 - heap pools
 - See heap pools

heap_id 0
See also heap storage, initial heap
segment
See initial heap

I

I/O
See input/output
IBM Data Facility Sort (DFSORT)
See DFSORT (Data Facility Sort)
Information Management System (IMS)
See IMS (Information Management System)
init_main
See preinitialization facility,
CELQPIPI(init_main)
init_sub
See preinitialization facility,
CELQPIPI(init_sub)
init_sub_dp
See preinitialization facility,
CELQPIPI(init_sub_dp)
initial heap segment
See heap storage, initial heap segment
initializing
See preinitialization facility
Interactive System Productivity Facility (ISPF)
See ISPF (Interactive System Productivity Facility)
interlanguage communication (ILC)
See also external data
See ILC (interlanguage communication)
interrupts
See program interrupts

K

keyboard 199

L

link pack area (LPA)
See LPA (link pack area)

M

mainframe
education xv
math services
about 145
message handling
See ISI (instance specific information)
Multiple Virtual System (MVS)
See MVS (Multiple Virtual System)
MVS (Multiple Virtual System)
binding for
See binding, for MVS
using system programming facilities
See system programming facility, C

N

nested enclave
See also enclave, nested
See SVC LINK
NLS (national language support)
See national language support (NLS)
Notices 203

P

parameter
See omitted parameter
PIPI table
See preinitialization facility, PIPI table
POSIX
position in Language Environment environment 4
preinitialization facility
service routines for
See @LOAD, @DELETE,
@GETSTORE, @FREESTORE,
@EXCEPRTN, @MSGRTN
program
binding using c89 41
placing MVS load modules in the HFS 43
running under z/OS UNIX 42

R

run-time environment, introduction 4

S

shortcut keys 199
SIGABRT
See abort() function
signal()
See condition handling
start_seq
See preinitialization facility,
CELQPIPI(start_seq)
syntax diagrams
how to read xii

T

Time Sharing Option (TSO)
See TSO (Time Sharing Option)
token, condition
See condition token

V

variables 14
exported 14

Z

z/OS Basic Skills information center xv
z/OS UNIX
position in Language Environment environment 4



Product Number: 5694-A01

Printed in USA

SA22-7569-06

