z/OS
Version 2 Release 3

Language Environment
Programming Guide

IBM
Part 1. Creating applications with Language Environment.................................1

Chapter 1. Introduction to Language Environment..................................................3
  Components of Language Environment.................................................................3
  Common runtime environment of Language Environment....................................4

Chapter 2. Preparing to link-edit and run under Language Environment...................7
  Understanding Language Environment library routines .......................................7
  Planning to link-edit and run....................................................................................7
  Link-editing single-language applications............................................................9
  Link-editing ILC applications.................................................................................9
  Downward compatibility considerations...............................................................9
  Checking which runtime options are in effect........................................................11
  HLL compatibility considerations.........................................................................12
  C/C++ AMODE/RMODE considerations...............................................................12
  COBOL considerations..........................................................................................13
    Replacing COBOL library routines in a COBOL load module...............................13
    Using Language Environment resident routines for callable services.................13
  Fortran considerations..........................................................................................14
    Replacing Fortran runtime library modules in a Fortran executable program........14
    Using the Language Environment library module replacement tool..................14
    Resolving static common block name conflicts.................................................15
    Resolving library module name conflicts between Fortran and C....................15
  PL/I considerations...............................................................................................17
    Link-editing PL/I subroutines for later use.......................................................22
    Replacing PL/I library routines in an OS PL/I executable program....................22
    Link-editing fetchable executable programs....................................................22
    PL/I link-time considerations.........................................................................23
    Fetching modules with different AMODEs.........................................................23

Chapter 3. Using Extra Performance Linkage (XPLINK).........................................25
Basic link-editing and running under TSO/E............................................................... 71
Accepting the default runtime options................................................................. 71
Overriding the default runtime options............................................................. 71
Specifying runtime options with the CEEOPTS DD card................................. 71
Link-editing and running...................................................................................... 72
Link-editing your application using the LINK command................................. 72
Using CMOD CLIST to invoke the TSO/E LINK command............................. 73
Using the CALL command to run your application........................................... 74
TSO/E parameter list format............................................................................. 75
Loading and running using the LOADGO command....................................... 75
Allocating data sets under TSO/E................................................................. 76
Example of using LOADGO............................................................................. 76
Link-edit and loader options.......................................................................... 77
Using the iconv utility and ICONV CLIST for C/C++................................. 77
Using the genxlt utility and GENXLT CLIST for C/C++................................. 78
Running your application under TSO/E......................................................... 78

Chapter 7. Creating and executing programs using z/OS UNIX System Services........ 79
Basic link-editing and running C/C++ applications under............................ 79
Invoking a shell from TSO/E........................................................................... 79
Using the z/OS UNIX c89 utility to link-edit and create executable files......... 80
Running z/OS UNIX C/C++ application programs........................................ 80
z/OS UNIX application program environments............................................. 80
Placing an MVS application executable program in the file system................ 81
Restriction on using 24-Bit AMODE programs.............................................. 81
Running an MVS executable program from a z/OS UNIX shell...................... 81
Running POSIX-enabled programs............................................................... 81
Running COBOL programs under z/OS UNIX.......................................... 83
Basic link-editing and running PL/I routines under z/OS UNIX with POSIX(ON) 84
Basic link-editing and running PL/I MTF applications under z/OS UNIX.... 86

Chapter 8. Using IBM-supplied cataloged procedures....................................... 87
Invoking cataloged procedures...................................................................... 87
Step names in cataloged procedures............................................................. 87
Unit names in cataloged procedures............................................................. 88
Data set names in cataloged procedures....................................................... 88
IBM-supplied cataloged procedures............................................................ 88
CEEWG — Load and run a Language Environment conforming non XPLINK program........ 91
CEEWL — Link a Language Environment conforming non XPLINK program...... 92
CEEWLG — Link and run a Language Environment conforming non-XPLINK program...... 92
CEEXR — Load and run a Language Environment conforming XPLINK program..... 93
CEEXL — Link-edit a Language Environment conforming XPLINK program........ 93
CEEXLR — Link and run a Language Environment conforming XPLINK program.... 94
AFHWL — Link a program written in Fortran............................................... 95
AFHWLG — Link and run a program written in Fortran............................... 96
AFHWN — Resolving name conflicts between C and Fortran....................... 96
Modifying cataloged procedures...................................................................... 97
Overriding and adding to EXEC statements.................................................. 97
Overriding and adding DD statements.......................................................... 97
Overriding generic link-edit procedures for constructed reentrant programs..... 99

Chapter 9. Using runtime options................................................................. 101
Methods available for specifying runtime options........................................ 101
Order of precedence....................................................................................... 103
Order of precedence examples...................................................................... 104
Specifying suboptions in runtime options..................................................... 104
Specifying runtime options and program arguments................................... 104
Creating application-specific runtime option defaults with CEEOPT................ 105
Part 2. Preparing an application to run with Language Environment ....................... 111

Chapter 10. Using Language Environment parameter list formats........................................ 113
  Argument lists and parameter lists.................................................................................. 113
  Passing arguments between routines.......................................................................... 114
  Preparing your main routine to receive parameters...................................................... 116
    PL/I argument passing considerations ..................................................................... 119

Chapter 11. Making your application reentrant.............................................................. 121
  Making your C/C++ program reentrant........................................................................ 121
    Natural reentrancy..................................................................................................... 121
    Constructed reentrancy............................................................................................ 121
    Generating a reentrant program executable for C or C++........................................ 122
  Making your COBOL program reentrant.................................................................... 122
  Making your Fortran program reentrant....................................................................... 122
  Making your PL/I program reentrant.......................................................................... 123
  Installing a reentrant load module.............................................................................. 123

Part 3. Language Environment concepts, services, and models................................. 125

Chapter 12. Initialization and termination under Language Environment.......................... 127
  The basics of initialization and termination................................................................. 127
  Language Environment initialization.......................................................................... 128
    What happens during initialization........................................................................ 128
  Language Environment termination........................................................................... 130
    What causes termination........................................................................................ 130
    What happens during termination......................................................................... 130
  Managing return codes in Language Environment.................................................... 132
    How the Language Environment enclave return code is calculated..................... 132
    Setting and altering user return codes.................................................................... 133
  Termination behavior for unhandled conditions........................................................ 135
    Determining the abend code.................................................................................... 136

Chapter 13. Program management model....................................................................... 139
  Model terminology for Language Environment program management................... 139
    Language Environment terms and their HLL equivalents......................................... 139
    Terminology for data............................................................................................... 140
  Processes................................................................................................................... 141
  Enclaves.................................................................................................................... 141
    The enclave defines the scope of language semantics............................................ 141
    Additional enclave characteristics.......................................................................... 142
  Threads...................................................................................................................... 142
  The full Language Environment program management model.................................. 142
  Mapping the POSIX program management model to the Language Environment program management model................................................................. 143
  Key POSIX program entities and Language Environment counterparts.................... 143
  Scope of POSIX semantics....................................................................................... 144
Chapter 24. Math services.......................................................................................................................343
What Language Environment math services does...............................................................................343
Related services....................................................................................................................................343
Call interface to math services.............................................................................................................345
Parameter types: parm1 type and parm2 type.....................................................................................345
Examples of calling math services.......................................................................................................346
Calling CEESSLOG in C and C++.......................................................................................................346
Calling CEESSLOG in COBOL.............................................................................................................347
Calling CEESSLOG in PL/I..................................................................................................................348

Part 4. Using interfaces to other products..........................................................................................349

Chapter 25. Running applications under CICS....................................................................................351
Terminology used in the Language Environment program model.....................................................351
CICS region...........................................................................................................................................351
CICS transaction....................................................................................................................................351
CICS run unit..........................................................................................................................................351
Running Language Environment applications under CICS...............................................................352
Developing an application under CICS................................................................................................352
PL/I coding considerations under CICS............................................................................................353
Assembler considerations....................................................................................................................353
Link-edit considerations under CICS..................................................................................................353
CICS processing program table (PPT) considerations........................................................................354
Specifying runtime options under CICS..............................................................................................354
Accessing DLI databases from CICS....................................................................................................357
Using callable services under CICS.....................................................................................................357
OS/VS COBOL compatibility considerations under CICS.................................................................357
Using math services in PL/I under CICS.............................................................................................357
Coding program termination in PL/I under CICS...............................................................................358
Storage management under CICS.........................................................................................................358
CICS short-on-storage condition..........................................................................................................358
CICS storage protect facility..................................................................................................................358
PL/I storage considerations under CICS.............................................................................................358
Condition handling under CICS...........................................................................................................359
PL/I considerations for using the CICS HANDLE ABEND command................................................359
Effect of the CICS HANDLE ABEND command..............................................................................360
Effect of the CICS HANDLE CONDITION and CICS HANDLE AID................................................360
Restrictions on user-written condition handlers under CICS..............................................................360
CICS transaction abend codes..............................................................................................................361
Using the CBLPSHPOP runtime option under CICS.........................................................................361
Restrictions on assembler user exits under CICS................................................................................361
Ensuring transaction rollback under CICS...........................................................................................362
Runtime output under CICS..................................................................................................................362
Message handling under CICS.............................................................................................................362
Dump services under CICS...................................................................................................................363
Support for calls within the same HLL under CICS.............................................................................363
C.........................................................................................................................................................363
C++....................................................................................................................................................363
COBOL..................................................................................................................................................363
Appendix A. Prelinking an application................................................................. 481
   Which programs need to be prelinked............................................................... 481
   What the prelinker does..................................................................................... 482
   Prelinking process.............................................................................................. 482
   References to currently unresolved symbols (unresolved external references)... 484
   Processing the prelinker automatic library call.................................................. 485
   Language Environment prelinker map.............................................................. 485
   Control statement processing............................................................................. 488
   IMPORT control statement............................................................................... 488
   INCLUDE control statement............................................................................. 488
   LIBRARY control statement............................................................................ 489
   RENAME control statement............................................................................. 489
   Mapping L-Names to S-Names.......................................................................... 490
   Starting the prelinker under batch and TSO/E................................................... 491
   Under batch....................................................................................................... 491
   Under TSO/E.................................................................................................... 492
   Using the CXXBIND EXEC under TSO/E.......................................................... 494
   Using the CXXMOD EXEC under TSO/E............................................................ 494
   Prelinker options............................................................................................... 496

Appendix B. EXEC DLI and CALL IMS Interfaces............................................... 499

Appendix C. Guidelines for writing callable services.......................................... 501

Appendix D. Operating system and subsystem parameter list formats............... 503
   C and C++ parameter passing considerations.................................................. 503
   C PLIST and EXECOPS interactions................................................................. 505
   C++ PLIST and EXECOPS interactions.............................................................. 507
   Case sensitivity under TSO................................................................................ 509
   Parameter passing considerations with XPLINK C and C++............................ 509
   COBOL parameter passing considerations..................................................... 509
   PL/I main procedure parameter passing considerations.................................. 511

Appendix E. Object library utility................................................................. 513
   Creating an object library.................................................................................. 513
   Under batch....................................................................................................... 513
   Under TSO....................................................................................................... 514
   Object library utility map.................................................................................. 515

Appendix F. Using the systems programming environment............................... 517
   Building freestanding applications.................................................................. 517
   Special considerations for reentrant modules.................................................. 518
   Notes.................................................................................................................. 519
   Building system exit routines.......................................................................... 520
   Building persistent C environments.................................................................. 520
   Building user-server environments................................................................... 520
   Summary of types.............................................................................................. 520

Appendix G. Sort and merge considerations..................................................... 523
   Invoking DFSORT directly................................................................................ 523
   Using the COBOL SORT and MERGE verbs.................................................... 523
   User exit considerations.................................................................................... 524
   Condition handling considerations.................................................................... 524
   Using the PL/I PLISRttx interface..................................................................... 525
   User exit considerations.................................................................................... 525
Appendix H. Running COBOL programs under ISPF............................................. 527
Appendix I. Language Environment macros................................................ 529
Appendix J. PL/I macros that activate variables............................................ 531
Appendix K. Accessibility................................................................................ 533
Accessibility features.................................................................................. 533
Consult assistive technologies...................................................................... 533
Keyboard navigation of the user interface.................................................... 533
Dotted decimal syntax diagrams............................................................... 533
Notices......................................................................................................... 537
Terms and conditions for product documentation........................................ 538
IBM Online Privacy Statement................................................................. 539
Policy for unsupported hardware............................................................... 539
Minimum supported hardware.................................................................... 540
Programming interface information........................................................... 540
Trademarks.................................................................................................. 540
Index.......................................................................................................... 541
List of Figures

1. Components of Language Environment........................................................................................................3
2. The common runtime environment of Language Environment........................................................................5
3. Replacing VS FORTRAN runtime library modules under batch, using CEEWL............................................ 14
4. Replacing VS FORTRAN runtime library modules under TSO/E, using a CLIST........................................ 15
5. Changing conflicting names in an executable program under MVS........................................................... 18
6. Changing conflicting names in several executable programs under MVS.................................................. 18
7. Changing conflicting names in an executable program under TSO/E........................................................18
8. Changing conflicting names in several executable programs under TSO/E.................................................19
9. Changing conflicting names in multiple executable programs under MVS...............................................19
10. Changing conflicting names in multiple executable programs under TSO/E...........................................19
11. Replacing Fortran routines with Language Environment routines under MVS............................................20
12. Replacing Fortran routines with Language Environment routines under TSO/E........................................20
13. Relink-editing an executable program to resolve conflicting names under batch....................................21
14. Relink-editing an executable program to resolve conflicting names under TSO/E....................................21
15. Example of link-editing a fetchable executable program............................................................................23
16. Standard stack storage model....................................................................................................................27
17. XPLINK stack storage model.....................................................................................................................28
18. XPLINK stack frame layout.......................................................................................................................29
19. Using #pragma export to create a DLL executable module named BASICIO.............................................46
20. Using #pragma export to create a DLL executable module TRIANGLE....................................................46
21. Using _export to create DLL executable module TRIANGLE...................................................................47
22. Using Language Environment macros to create an assembler DLL executable named ADLlBEV2.........48
23. COBOL DLL application calling a COBOL DLL.......................................................................................52
24. Assembler DLL application calling an assembler DLL..............................................................................53
25. Summary of DLL and DLL application preparation and usage..............................................................54
26. Accepting the default runtime options under batch................................................................................58
27. Overriding the default runtime options under batch................................................................................58
28. Overriding the default runtime options for COBOL................................................................................58
29. Basic batch link-edit processing................................................................................................................60
30. Creating a non-XPLINK executable program under batch........................................................................63
31. Creating an XPLINK executable program under batch............................................................................63
32. Using the INCLUDE linkage editor control statement.............................................................................64
33. Using the LIBRARY linkage editor control statement.............................................................................64
34. Basic loader processing................................................................................................................................66
35. JCL for creating an executable program..................................................................................................68
36. Cataloged procedure CEEWG, which loads and runs a program written in any Language Environment-conforming HLL.................................................................92
37. Cataloged procedure CEEWL, which link-edits a program written in any Language Environment-conforming HLL.................................................................92
80. q_data structure for arithmetic program interruption conditions..............................................248
81. q_data structure for math and bit manipulation routines..............................................................251
82. Format of a q_data descriptor..................................................................................................254
83. Example of a message source file.........................................................................................258
84. Example of a message module table with one language.........................................................267
85. Example of a message module table with two languages.......................................................267
86. Example of assigning values to message inserts.....................................................................269
87. Directing output messages.......................................................................................................277
88. Performing calculations on dates.............................................................................................287
89. Default century window...........................................................................................................288
90. Using CEESCN to change the century window........................................................................288
91. z/OS XL C/C++ routine with a call to CEEFMDT....................................................................340
92. COBOL program with a call to CEEFMDT.............................................................................341
93. PL/I routine with a call to CEEFMDT......................................................................................342
94. C/C++ call to CEESLOG — Logarithm base e.........................................................................346
95. Call to CEESLOG — Logarithm base e in COBOL.................................................................347
96. Call to CEESLOG — Logarithm base e in PL/I......................................................................348
97. Format of messages sent to CESE..........................................................................................362
98. Location of user exits................................................................................................................376
99. Interface for the CEEBXITA assembler user exit.................................................................380
100. CEEAUE_FLAGS format........................................................................................................381
101. Exit_list and hook_exit control blocks...............................................................................389
102. Example calling routine named Bif5 with no parameters....................................................419
103. Example calling routine named Bif5 passing 5 integer parameters.....................................419
104. Example of a simple main assembler routine.........................................................................421
105. Sample invocation of a callable service from assembler......................................................424
106. Issuing an ATTACH to Language Environment-conforming routines.....................................426
107. A dynamically-called COBOL program that dynamically calls another COBOL program....428
108. Format of service routine vector..........................................................................................459
109. Basic prelinker and linkage editor processing........................................................................484
110. Example of prelinking under batch.......................................................................................494
111. Example of prelinking under TSO/E....................................................................................494
112. Alternate C/C++ parameter passing styles............................................................................503
113. Accessing parameters using macros __R1 and __osplist.......................................................504
114. Examples of casting and dereferencing..................................................................................504
115. Object library utility map........................................................................................................515
116. Specifying alternate initialization at link-edit time.................................................................517
117. Simple freestanding routine..................................................................................................518
118. Link-edit control statements used to build a freestanding MVS routine..............................518
119. Compile and link by using the cataloged procedure EDCCL...............................................518
120. Sample reentrant freestanding routine..................................................................................519
121. Building and running a reentrant freestanding MVS routine................................................519
List of Tables

1. How to use z/OS Language Environment publications................................................................. xxiv
2. Syntax examples................................................................................................................................... xxvi
3. Prerequisite OS/390 release level for the various compilers that support downward compatibility......10
4. Fortran and C library routine names that are identical...................................................................... 15
5. Conflicting names per product and release........................................................................................ 16
6. Decision table for name conflict resolution........................................................................................ 17
7. Comparing non-XPLINK and XPLINK register conventions.............................................................. 31
8. Required data sets used for link-editing.............................................................................................. 61
9. Optional data sets used for link-editing............................................................................................... 62
10. Selected link-edit options.................................................................................................................. 65
11. Selected loader options...................................................................................................................... 67
12. Standard loader data sets.................................................................................................................... 68
13. CMOD calls........................................................................................................................................ 74
14. Selected loader options...................................................................................................................... 77
15. IBM-supplied cataloged procedures................................................................................................... 88
16. Formats for specifying runtime options and program arguments................................................... 104
17. Semantic terms and methods for passing arguments in Language Environment............................ 114
18. Default passing style per HLL........................................................................................................... 115
19. Coding a main routine to receive an inbound parameter list in TSO/E.............................................. 116
20. Coding a main routine to receive an inbound parameter list in IMS............................................... 118
21. Coding a main routine to receive an inbound parameter list in CICS.............................................. 118
22. Coding a main routine to receive an inbound parameter list in MVS.............................................. 119
23. Fortran reentrancy separation tool and Language Environment cataloged procedures................... 122
24. Return code modifiers used by Language Environment to determine enclave return codes............ 134
25. Summary of enclave reason codes................................................................................................... 135
26. Termination behavior for unhandled conditions of severity 2 or greater....................................... 136
27. Abend codes used by Language Environment when the Assembler user exit requests an abend..... 136
28. Abend code values used by Language Environment with ABTERMENC(ABEND)......................... 136
29. Program interrupt abend and reason codes in a non-CICS environment........................................ 137
30. Usage of stack and heap storage by Language Environment-conforming languages.................... 147
31. Runtime options and functions......................................................................................................... 147
32. Callable services options and functions............................................................................................ 148
33. Heap IDs recognized by Language Environment heap manager.................................................... 152
34. Default responses to unhandled conditions...................................................................................... 174
35. T_I_U condition representation......................................................................................................... 175
36. T_I_S condition representation......................................................................................................... 176
37. C conditions and default system actions........................................................................................... 184
38. Mapping of S/370 exceptions to C signals....................................................................................... 190
39. Mapping of abend signals to C signals............................................................................................. 190
83. Parameters for Language Environment condition handler.................................................................464
84. Return and reason codes from the Language Environment condition handler.....................................464
85. Return and reason codes for the @MSGRTN service............................................................................465
86. Handling conditions in child enclaves................................................................................................472
87. Unhandled condition behavior in a C, C++, or assembler child enclave.............................................472
88. Unhandled condition behavior in a COBOL child enclave..................................................................473
89. Unhandled condition behavior in a Fortran or PL/I child enclave.........................................................473
90. Unhandled condition behavior in a system()-created child enclave....................................................474
91. Unhandled condition behavior in a child enclave that contains a PL/I fetchable main..........................475
92. Determining the command-line equivalent..........................................................................................477
93. Determining the order of runtime options and program arguments....................................................477
94. Prelinker options..................................................................................................................................497
95. IMS and CICS support of user interfaces to DL/I databases..................................................................499
96. Interactions of C PLIST and EXECOPS..............................................................................................506
97. Interactions of C/C++ PLIST and EXECOPS (compiler options)..........................................................508
98. Case sensitivity of arguments under TSO............................................................................................509
99. Interactions of SYSTEM and NOEXECOPS.....................................................................................511
100. Summary of types..............................................................................................................................520
101. DFSORT exit called as a function of a PLISRTx interface call............................................................525
102. Variables activated by PL/I macros....................................................................................................531
IBM® z/OS Language Environment® (also called Language Environment) provides common services and language-specific routines in a single runtime environment for C, C++, COBOL, Fortran (z/OS only; no support for z/OS UNIX or CICS®), PL/I, 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 that are generated with the following IBM compiler products:

- z/OS XL C/C++ (feature of z/OS)
- z/OS C/C++
- OS/390® C/C++
- C/C++ for MVS/ESA
- C/C++ for z/VM®
- XL C/C++ for z/VM
- AD/Cycle C/370™
- VisualAge® for Java™, Enterprise Edition for OS/390
- Enterprise COBOL for z/OS
- Enterprise COBOL for z/OS and OS/390
- COBOL for OS/390 and VM
- COBOL for MVS™ & VM (formerly COBOL/380)
- Enterprise PL/I for z/OS
- Enterprise PL/I for z/OS and OS/390
- VisualAge PL/I
- PL/I for MVS and VM (formerly PL/I MVS & VM)
- VS FORTRAN and FORTRAN IV (in compatibility mode)

Although not all compilers listed are currently supported, Language Environment supports the compiled objects that they created.

Language Environment supports, but is not required for, an interactive debug tool for debugging applications in your native z/OS environment.

IBM z/OS Debugger is also available as a stand-alone product. For more information, see IBM z/OS Debugger (developer.ibm.com/mainframe/products/ibm-zos-debugger).

Language Environment supports, but is not required for, VS FORTRAN Version 2 compiled code (z/OS only).

Language Environment consists of the common execution library (CEL) and the runtime libraries for C/C++, COBOL, Fortran, and PL/I.

For more information about VisualAge for Java, Enterprise Edition for OS/390, program number 5655-JAV, see the product documentation.
• Manage the runtime 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 on page xxiv.

<table>
<thead>
<tr>
<th>To …</th>
<th>Use …</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluate Language Environment</td>
<td>z/OS Language Environment Concepts Guide</td>
</tr>
<tr>
<td>Plan for Language Environment</td>
<td>z/OS Language Environment Concepts Guide</td>
</tr>
<tr>
<td></td>
<td>z/OS Language Environment Runtime Application Migration Guide</td>
</tr>
<tr>
<td>Install Language Environment</td>
<td>z/OS Program Directory in the z/OS Internet library (<a href="http://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary">www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary</a>)</td>
</tr>
<tr>
<td>Customize Language Environment</td>
<td>z/OS Language Environment Customization</td>
</tr>
<tr>
<td>Understand Language Environment program models and concepts</td>
<td>z/OS Language Environment Concepts Guide</td>
</tr>
<tr>
<td></td>
<td>z/OS Language Environment Programming Guide</td>
</tr>
<tr>
<td></td>
<td>z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode</td>
</tr>
<tr>
<td>Find syntax for Language Environment runtime options and callable services</td>
<td>z/OS Language Environment Programming Reference</td>
</tr>
<tr>
<td>Develop applications that run with Language Environment</td>
<td>z/OS Language Environment Programming Guide and your language programming guide</td>
</tr>
<tr>
<td>Debug applications that run with Language Environment, diagnose problems with Language Environment</td>
<td>z/OS Language Environment Debugging Guide</td>
</tr>
<tr>
<td>Get details on runtime messages</td>
<td>z/OS Language Environment Runtime Messages</td>
</tr>
<tr>
<td>Develop interlanguage communication (ILC) applications</td>
<td>z/OS Language Environment Writing Interlanguage Communication Applications and your language programming guide</td>
</tr>
<tr>
<td>Migrate applications to Language Environment</td>
<td>z/OS Language Environment Runtime Application Migration Guide and the migration guide for each Language Environment-enabled language</td>
</tr>
</tbody>
</table>
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 IBM Knowledge Center using a screen reader, syntax diagrams are provided in dotted decimal format.

Symbols

The following symbols may be displayed in syntax diagrams:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>►►───</td>
<td>Indicates the beginning of the syntax diagram.</td>
</tr>
<tr>
<td>───►</td>
<td>Indicates that the syntax diagram is continued to the next line.</td>
</tr>
<tr>
<td>◄───</td>
<td>Indicates that the syntax is continued from the previous line.</td>
</tr>
<tr>
<td>───◄</td>
<td>Indicates the end of the syntax diagram.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required</strong></td>
</tr>
<tr>
<td>Required items are displayed on the main path of the horizontal line.</td>
</tr>
<tr>
<td><strong>Optional</strong></td>
</tr>
<tr>
<td>Optional items are displayed below the main path of the horizontal line.</td>
</tr>
<tr>
<td><strong>Default</strong></td>
</tr>
<tr>
<td>Default items are displayed above the main path of the horizontal line.</td>
</tr>
</tbody>
</table>
## Syntax examples

The following table provides syntax examples.

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required item.</td>
<td><img src="image1" alt="Required Item Example" /></td>
</tr>
<tr>
<td>Required items appear on the main path of the horizontal line. You must specify these items.</td>
<td></td>
</tr>
<tr>
<td>Required choice.</td>
<td><img src="image2" alt="Required Choice Example" /></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Optional item.</td>
<td><img src="image3" alt="Optional Item Example" /></td>
</tr>
<tr>
<td>Optional items appear below the main path of the horizontal line.</td>
<td></td>
</tr>
<tr>
<td>Optional choice.</td>
<td><img src="image4" alt="Optional Choice Example" /></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Default.</td>
<td><img src="image5" alt="Default Example" /></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Variable.</td>
<td><img src="image6" alt="Variable Example" /></td>
</tr>
<tr>
<td>Variables appear in lowercase italics. They represent names or values.</td>
<td></td>
</tr>
<tr>
<td>Repeatable item.</td>
<td><img src="image7" alt="Repeatable Item Example" /></td>
</tr>
<tr>
<td>An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.</td>
<td></td>
</tr>
<tr>
<td>A character within the arrow means you must separate repeated items with that character.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
This Programming Guide

You should be familiar with the Language Environment product and one or more of the supported Language Environment-conforming high-level languages. The term C/C++ is used generically to refer to information that applies to both C and C++.

Previous versions of the Language Environment-conforming language products provided their own environment and services for running applications, and their associated application programming guides including information on how to link-edit and run applications. Language Environment now provides the runtime support required to run applications compiled under all of the Language Environment-conforming HLLs, as well as the facility for interlanguage communication between supported languages.

For application programming, you will need to use the following books:

- This book contains information about linking, running, and using services within the Language Environment environment, the Language Environment program management 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.
- z/OS Language Environment Writing Interlanguage Communication Applications provides information to help you create and run interlanguage communication (ILC) applications.

This book is organized as follows:

- Part 1 includes a basic introduction to Language Environment. It also describes linking, loading, and running under each of the supported operating systems, as well as using IBM-supplied cataloged procedures, Language Environment runtime options, and Language Environment callable services.
- 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 management model, storage, condition handling, messages, callable services, and math services.
- Part 4 explains using interfaces to other products such as CICS, Db2®, and IMS™.
- Part 5 addresses specialized programming tasks, such as using runtime user exits, assembler considerations, preinitialization services, and using nested enclaves.
- The various appendixes describe interfaces to subsystems, writing callable services, using parameter list formats, prelinking, using the C object library, systems programming environments, sort and merge considerations, ISPF, and Language Environment macros.

Table 2: Syntax examples (continued)

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fragment.</td>
<td>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.</td>
</tr>
</tbody>
</table>
How to send your comments to IBM

We invite you to submit comments about the z/OS product documentation. Your valuable feedback helps to ensure accurate and high-quality information.

Important: If your comment regards a technical question or problem, see instead “If you have a technical problem” on page xxix.

Submit your feedback by using the appropriate method for your type of comment or question:

Feedback on z/OS function
If your comment or question is about z/OS itself, submit a request through the IBM RFE Community (www.ibm.com/developerworks/rfe/).

Feedback on IBM Knowledge Center function
If your comment or question is about the IBM Knowledge Center functionality, for example search capabilities or how to arrange the browser view, send a detailed email to IBM Knowledge Center Support at ibmkc@us.ibm.com.

Feedback on the z/OS product documentation and content
If your comment is about the information that is provided in the z/OS product documentation library, send a detailed email to mhvrcfs@us.ibm.com. We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

To help us better process your submission, include the following information:

• Your name, company/university/institution name, and email address
• The following deliverable title and order number: z/OS Language Environment Programming Guide, SA38-0682-30
• The section title of the specific information to which your comment relates
• The text of your comment.

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

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

If you have a technical problem

If you have a technical problem or question, do not use the feedback methods that are provided for sending documentation comments. Instead, take one or more of the following actions:

• Go to the IBM Support Portal (support.ibm.com).
• Contact your IBM service representative.
• Call IBM technical support.
Summary of changes

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

Summary of changes for Language Environment for z/OS Version 2 Release 3 (V2R3)

New

• To validate stack guards in stacks, STKPROT= was added to these macros:
  – “CEEENTRY macro—Generate a Language-Environment-conforming prolog” on page 400
  – “CEEPPA macro—Generate a PPA” on page 404.
  – “CEEXPIT” on page 431

Changed

• Various updates were made to replace IBM Debug Tool for z/OS with IBM z/OS Debugger.

Summary of changes for Language Environment for z/OS Version 2 Release 2 (V2R2)

New

• Two new keywords were added to the CEEPPA macro: VRSMASK and VRSLOCR. See “CEEPPA macro—Generate a PPA” on page 404.
• A new usage note was added to “CEEFETCH macro—Dynamically load a routine” on page 409. See usage note 12.

Changed

• The origin for Masked in Table 37 on page 184 was corrected.
• “C++ PLIST and EXECOPS interactions” on page 507 was updated.

Deleted

• No content was removed from this information.

Summary of changes for Language Environment for z/OS Version 2 Release 1 (V2R1) as updated February, 2015

New

• In support of vector registers, two fields (VRSMASK and VRSLOCR) were added to the CEEPPA macro. See “CEEPPA macro—Generate a PPA” on page 404.
z/OS Version 2 Release 1 summary of changes

See the Version 2 Release 1 (V2R1) versions of the following publications for all enhancements related to z/OS V2R1:

- z/OS Migration
- z/OS Planning for Installation
- z/OS Summary of Message and Interface Changes
- z/OS Introduction and Release Guide
Part 1. Creating applications with Language Environment

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

**Note:** The terms having to do with linking (bind, binding, link, link-edit, and so on) refer to the process of creating an executable program from object modules (the output that is 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. For information that is specific to the linkage editor, see z/OS MVS Program Management: User’s Guide and Reference and z/OS MVS Program Management: Advanced Facilities.

If you have an application that contains interlanguage calls, you might need to relink-edit it to take advantage of the Language Environment ILC support. See z/OS Language Environment Runtime Application Migration Guide for more information.
Chapter 1. Introduction to Language Environment

Language Environment provides a common runtime environment for IBM versions of certain high-level languages (HLLs), namely, C, C++, COBOL, Fortran, and PL/I, in which you can run existing applications written in previous versions of these languages as well as in the current versions. Before Language Environment, each of the HLLs had to provide a separate runtime environment.

Language Environment combines essential and commonly used runtime services, such as routines for runtime message handling, condition handling, storage management, date and time services, and math functions, and makes them available through a set of interfaces that are consistent across programming languages. With Language Environment, you can use one runtime environment for your applications, regardless of the application's programming language or system resource needs because most system dependencies have been removed.

Language Environment provides compatible support for existing HLL applications; most existing single-language applications can run under Language Environment without being recompiled or relink-edited. POSIX-conforming C applications can use all Language Environment services.

Components of Language Environment

As Figure 1 on page 3 shows, Language Environment consists of the following components:

- Basic routines that support starting and stopping programs, allocating storage, communicating with programs written in different languages, and indicating and handling error conditions.
- Common library services, such as math services and date and time services, that are commonly needed by programs running on the system. These functions are supported through a library of callable services.
- Language-specific portions of the common runtime library.

The following IBM language compilers currently participate in this release:

- z/OS XL C/C++
- C/C++ Compiler for MVS/ESA
- AD/Cycle C/370 Compiler
Common runtime environment of Language Environment

The common runtime environment of Language Environment illustrates the common environment that Language Environment creates.
Figure 2: The common runtime environment of Language Environment
Chapter 2. Preparing to link-edit and run under Language Environment

This topic discusses what you need to know before link-editing and running applications under Language Environment. After Language Environment is installed on your system, you should run an existing application under Language Environment. Although you may need to link-edit to different libraries, the procedure is similar to that used in pre-Language Environment versions of C, COBOL, Fortran, or PL/I.

To help you get started, this topic describes the following common link-edit and run tasks, which you might want to try before reading further:

- Link-editing and running an existing object module and accepting the default Language Environment runtime options
- Link-editing and running an existing object module and specifying new Language Environment runtime options
- Calling a Language Environment service

Note that several Fortran and C library routines have identical names. This topic describes what you must do to resolve any potential conflicts in using these names.

This topic also describes basic tasks intended to help give you an idea of what running an application under Language Environment is like. It is not intended to illustrate every aspect of link-editing and running you might want to learn. Detailed instructions about link-editing and running existing and new applications under each supported operating system are provided in other topics in this information, and in z/OS MVS Program Management: User’s Guide and Reference and z/OS MVS Program Management: Advanced Facilities.

Understanding Language Environment library routines

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 initialization/termination routines and pointers to callable services. 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 application executable programs small. When maintaining dynamic library code, you need not relink-edit the application code except under special circumstances, such as when you use an earlier version of code.

The linkage editor converts an object module into an executable program and stores it in a library. The executable program can then be run from that library at any time. The link-edit process combines output from compilers, language translators, link-edit programs and control statements to produce an executable program (load module or program object) and stores it in a library (PDS, PDSE, or HFS). The executable program can then be run from that library. Either the program management binder or linkage editor can be used to perform the link-edit process. All of the services of the linkage editor can be performed by the binder. In addition, the binder provides additional functionality and usability improvements. See z/OS MVS Program Management: User’s Guide and Reference and z/OS MVS Program Management: Advanced Facilities for a complete discussion of services to create, load, modify, list, read, transport and copy executable programs.

Planning to link-edit and run

There are certain considerations that you must be aware of before link-editing and running applications under Language Environment.

Language Environment resident routines for non-XPLINK applications, including those for callable services, initialization, and termination, are located in the SCEELKED and SCEELKEX libraries. Language
Environment resident routines for XPLINK applications are located in the SCEEBIND and SCEEBND2 libraries. 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:

**SCEERUN**
A PDS that contains the runtime library routines that are needed during execution of applications written in C/C++, PL/I, COBOL and FORTRAN.

**SCEERUN2**
A PDSE that contains the runtime library routines that are needed during execution of applications written in C/C++ and COBOL.

**SCEELKED**
Contains Language Environment resident routines for non-XPLINK applications, including those for callable services, initialization, and termination. This includes language-specific callable services, such as those for the C/C++ runtime library. Only case-insensitive names of eight or less characters in length are contained in this library. **This library must be used only when link-editing a non-XPLINK program.**

**SCEELKEX**
Like SCEELKED, contains Language Environment resident routines for non-XPLINK applications. However, case-sensitive names that can be greater than eight characters in length are contained in this library. This allows symbols such as the C/C++ printf and pthread_create functions to be resolved without requiring the names to be uppercased, truncated, or mapped to another symbol. **This library must be used only when link-editing a non-XPLINK program.**

**SCEE0BJ**
Contains Language Environment resident definitions for non-XPLINK applications which may be required for z/OS UNIX (z/OS UNIX) programs, such as the definition of the external variable symbol environ. **SCEE0BJ must be used whenever link-editing a z/OS UNIX non-XPLINK program.**

**SCEECPP**
Contains Language Environment resident definitions for non-XPLINK applications which may be required for C++ programs, such as the definition of the new operator. **SCEECPP must be used whenever link-editing a program that includes any NOXPLINK-compiled C++ object modules.**

**SCEEBIND**
Contains Language Environment resident routines for XPLINK applications, but is deprecated and may not be supported in a future release. Use SCEEBND2 instead.

**SCEEBND2**
Contains all Language Environment resident routines for XPLINK applications. This one library replaces the four libraries of resident routines for non-XPLINK applications. For XPLINK, this one library is used wherever the four libraries of resident routines (SCEELKED, SCEELKEX, SCEEOBJ, SCEECPP) had been used. 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. **SCEEBND2 must be used only when link-editing an XPLINK program.**

**SCEELIB**
Contains side-decks for DLLs provided by Language Environment.

Many of the language-specific callable services available to XPLINK-compiled applications appear externally as DLL functions. See Chapter 4, “Building and using dynamic link libraries (DLLs),” on page 37 for information about DLLs. To resolve these references from XPLINK applications, a definition side deck must be included when link-editing the application. The SCEELIB library contains the following side decks in support of XPLINK:

- CELHS001 — side deck to resolve references to Language Environment services when link-editing an XPLINK application. This includes both Application Writer Interfaces (AWIs) and Compiler Writer Interfaces (CWIs).
  - For more information about AWIs, see z/OS Language Environment Programming Reference).
- For more information about CWIs, see z/OS Language Environment Vendor Interfaces.

The entries in this sidedeck replace the corresponding non-XPLINK resident routines in SCEELKED.

The AWI stubs also exist as executables in SCEERUN, which can be loaded and run from non-XPLINK applications. This technique cannot be used with XPLINK applications.

- CELHS003 — sidedeck to resolve references to callable services in the C/C++ runtime library when link-editing an XPLINK application. The entries in this sidedeck replace the corresponding non-XPLINK resident routines in SCEELKEX, SCEELKED, and SCEEOBJ.

- CELHSCPP — sidedeck to resolve references to C++ runtime library (RTL) definitions that might be required when link-editing an XPLINK application. The entries in this sidedeck replace the non-XPLINK resident routines in SCEECPP.

The functions in these sidedecks can be called from an XPLINK application. However, they cannot be used as the target of an explicit dllqueryfn() against the DLL.

**Link-editing single-language applications**

The default main entry point for a C, C++, or PL/I application is CEESTART (PLISTART for code compiled with OS PL/I); for a Fortran application, it is the name of the main routine. For COBOL, the main entry point for an application is determined in one of two ways:

- The name of the first object module presented to the link-edit process.
- Explicit specification of the entry point by providing a control statement to the link-edit process.

A copy of CEESTART resides in the Language Environment SCEELKED library. Do not explicitly include it in the link-edit process, even for Language Environment-conforming languages. The compilers generate CEESTART or references to it when necessary.

Although CEESTART is not used as an entry point by Language Environment-conforming assembler programs, it still must be resolved by the link-editor. To ensure this is possible, avoid using the NCAL link-editor option.

You must link-edit applications before you run them.

**Link-editing ILC applications**

When mixing languages within an application, presenting the desired main routine to the link-edit process first nominates it as the entry point. You can specify only one main routine.

To get Language Environment support in using pre-Language Environment C – COBOL ILC applications, you must relink-edit these applications to replace old HLL library routines with Language Environment routines. Relink-editing ILC applications of any language combination is usually required, with the following exceptions:

- Any PL/I – COBOL ILC applications relink-edited using the migration aid provided by OS PL/I Version 2 Release 3. (See the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735) for details.) The PTF numbers for the migration aid are UN76954 and UN76955.
- Any PL/I – C ILC applications.
- Any COBOL – C ILC applications relink-edited using the migration aid provided by the C/370 Version 2 Library. This migration aid was delivered in the fix for APAR PN74931.

For more information, see z/OS Language Environment Writing Interlanguage Communication Applications, z/OS Language Environment Runtime Application Migration Guide, and the migration guides for your primary HLL.

**Downward compatibility considerations**

As of OS/390 Version 2 Release 10, Language Environment provides downward compatibility support. Assuming that required programming guidelines and restrictions 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 OS/390 V2R10 or later (and Language Environment) on a development system where applications are coded, link edited, and tested, while using any supported lower release of OS/390 (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 OS/390 V2R10 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) (C/C++, COBOL, Fortran, PL/I) 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 products 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 OS/390 V2R10.
- 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 OS/390 V2R9 Language Environment, new support was added so that COBOL programs could dynamically call a reentrant C routine with constructed reentrancy without using #pragma(xxx, COBOL). This support is available on OS/390 V2R9 Language Environment or later, but is not available on prior releases.

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. (Prior releases of the compilers beyond those listed in the table are still supported by Language Environment, but do not provide downward compatibility. They only support upward compatibility.)

<table>
<thead>
<tr>
<th>Compiler product</th>
<th>OS/390 release level prerequisite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise COBOL for z/OS</td>
<td>OS/390 V2R10</td>
</tr>
</tbody>
</table>
Table 3: Prerequisite OS/390 release level for the various compilers that support downward compatibility (continued)

<table>
<thead>
<tr>
<th>Compiler product</th>
<th>OS/390 release level prerequisite</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL for MVS &amp; VM, V1R2</td>
<td>OS/390 V2R6</td>
</tr>
<tr>
<td>COBOL for OS/390 &amp; VM, V2R1</td>
<td>OS/390 V2R6</td>
</tr>
<tr>
<td>COBOL for OS/390 &amp; VM, V2R2</td>
<td>OS/390 V2R6</td>
</tr>
<tr>
<td>PL/I for MVS &amp; VM</td>
<td>OS/390 V2R6</td>
</tr>
<tr>
<td>OS PL/I 2.3</td>
<td>OS/390 V2R6</td>
</tr>
<tr>
<td>OS/390 C/C++ compiler</td>
<td>OS/390 V2R10</td>
</tr>
<tr>
<td>VS Fortran 2.6</td>
<td>OS/390 V2R6</td>
</tr>
</tbody>
</table>

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

- Options Processing: Whenever an application exploits Language Environment runtime 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.

**Note:** Starting with z/OS V1R8, the c89 utility is no longer by default passing COMPAT = CURRENT option to the program management binder. Program objects created by the c89 utility and native program management binder invocations will use the default COMPAT = MIN.

### Checking which runtime options are in effect

Using the Language Environment runtime option RPTOPTS, you can control whether a runtime options report is produced; with the Language Environment runtime option MSGFILE, you can control where report output is directed. RPTOPTS generates a report of all the runtime options that are in effect when your application begins to run. The IBM-supplied default for RPTOPTS is OFF, meaning a report is not generated when your application finishes running. If you override the default setting of RPTOPTS in any of the ways described below, a report is sent to the default location:

- On MVS, to the standard system data set SYSOUT. SYSOUT is dynamically allocated when needed, and is directed to whatever MSGCLASS you specified on the JOB card when you ran the application.
• Under z/OS UNIX, it goes to file descriptor 2.
• Under CICS, with RPTOPTS(ON), Language Environment writes the options report to the CESE queue when the transaction ends successfully.
• On TSO/E, to SYSOUT.

The MSGFILE runtime option is ignored under CICS, so it has no effect on where the options report goes.

If you want to change the options report destination, you can alter the default setting of the MSGFILE runtime option, which specifies where all runtime diagnostics and messages are written. For example, if you specify MSGFILE(OPTRPRT), the storage report is written to a file whose ddname is OPTRPRT. You need to allocate a data set for OPTRPRT under batch and TSO/E.

For the syntax of RPTOPTS and MSGFILE, see "z/OS Language Environment Programming Reference."
<table>
<thead>
<tr>
<th>Product</th>
<th>RMODE</th>
<th>AMODE</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ with Db2 R2.2</td>
<td>24 or ANY</td>
<td>31</td>
<td>All programs must use this AMODE and RMODE combination.</td>
</tr>
<tr>
<td>C/C++ with IMS/ESA® V3R1</td>
<td>24 or ANY</td>
<td>31</td>
<td>All programs must use the same AMODE and RMODE combination. There are no restrictions on IMS/ESA parameters.</td>
</tr>
<tr>
<td>One of the following:</td>
<td>24 or ANY</td>
<td>31</td>
<td>All programs must use the same AMODE and RMODE combination.</td>
</tr>
<tr>
<td>• C/C++ only</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• C/C++ with ISPF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• C/C++ with PL/I</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For information on AMODE switching, see z/OS XL C/C++ Runtime Library Reference.

**COBOL considerations**

This topic describes what you need to know if you link-edit or relink-edit a COBOL program with Language Environment.

**Replacing COBOL library routines in a COBOL load module**

Three sample jobs are provided in the SCEESAMP sample library that can be used to replace all OS/VS COBOL and VS COBOL II library routines in load modules containing OS/VS COBOL and VS COBOL II programs. These sample jobs are:

- IGZWRLKA to relink-edit a VS COBOL II load module with Language Environment.
- IGZWRLKB to relink-edit an OS/VS COBOL load module with Language Environment.
- IGZWRLKC to relink-edit a load module that contains both OS/VS COBOL programs and VS COBOL II programs with Language Environment.

For more information on relink-editing existing OS/VS COBOL and VS COBOL II load modules, see the Enterprise COBOL V4 Migration Guide in Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733).

**Using Language Environment resident routines for callable services**

For COBOL CALL literal statements, the compiler allows you to specify whether your program uses static or dynamic calls to Language Environment callable services (or other subroutines):

- When a COBOL program makes a static call to a Language Environment callable service, the Language Environment resident routine (a callable service stub) is link-edited with the program.
- When a COBOL program makes a dynamic call to a Language Environment callable service, the Language Environment resident routine is not link-edited with the program.

Only COBOL programs compiled with Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM or COBOL/370 can call Language Environment callable services.

**Note:** You can use dynamic calls from VS COBOL II programs to Language Environment Date/Time callable services. You cannot use dynamic call from VS COBOLL II programs to other Language Environment callable services. You cannot use static calls from VS COBOL II programs to any Language Environment callable services.

For more information about COBOL static and dynamic calls, see the appropriate version of the COBOL programming guide in the COBOL library at Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733).
Fortran considerations

This topic discusses what you need to know if you link-edit or relink-edit a Fortran program.

Replacing Fortran runtime library modules in a Fortran executable program

To relink-edit your existing executable program under Language Environment, you must replace Fortran runtime library modules in the executable program with the equivalent Language Environment routines. The Fortran library module replacement tool enables you to do this without having to have the object modules that make up the executable program. This is most useful when:

• You need to recompile some, but not all, of your own Fortran routines that are within one of your executable programs.
• You need to upgrade existing programs to contain the Language Environment runtime library modules. Language Environment data sets can be installed, but the changes are not reflected in your own executable programs unless you link-edit them again using the updated data sets.

You might have to use your original executable program rather than your object modules as linkage editor input because you don’t have all of your routines available in source form for recompilation or because you didn’t retain the object modules. A problem occurs when you use your previous executable programs as linkage editor input because the linkage editor retains the non-Language Environment modules that are in your original executable program while including others from the current SYSLIB input. The solution is to use the Fortran library module replacement tool as discussed in the following sections.

Using the Language Environment library module replacement tool

The Fortran library module replacement tool provides a set of linkage editor REPLACE statements to help you replace all of the runtime library modules when your input to the linkage editor is an existing executable program containing the library modules. The tool supports executable programs created by VS FORTRAN Version 1, VS FORTRAN Version 2, and Language Environment. The source file containing the Fortran library module replacement tool is member AFHWRLK in the CEE.SCEESAMP library.

In Figure 3 on page 14, the VS FORTRAN Version 2 runtime library modules in the executable program MYLMOD are replaced while retaining the compiled code, using the cataloged procedure CEEWL.

```
//RELINK     EXEC PROC=CEEWL,PGMLIB=MYPDS.LOAD,GOPGM=MYLMOD
//SYSPRINT   DD   SYSOUT=A
//SAMPLIB    DD   DSN=CEE.SCEESAMP,DISP=SHR
//USERLMOD   DD   DSN=MYPDS.LOAD,DISP=OLD
//LKED.SYSIN DD    *
   INCLUDE SAMPLIB(AFHWRLK)
   INCLUDE USERLMOD(MYLMOD)
   NAME    MYLMOD(R)
/*
```

Figure 3: Replacing VS FORTRAN runtime library modules under batch, using CEEWL

Figure 4 on page 15 shows how you can perform the same replacement under TSO/E, using a CLIST.
Resolving static common block name conflicts

It is possible for a Fortran static common block name in one program unit to be in conflict with a Fortran intrinsic function name in another program unit. (A conflict could arise, for example, if you used LOG as a common block name and invoked the LOG intrinsic function in a different program unit.) To avoid any such conflict, either rename the common block or recompile with VS FORTRAN Version 2.5 or later.

It is also possible that a Fortran static common block name could conflict with another language's library routine name. (A conflict could arise, for example, if you used GETS as a common block name, and also invoked C's `gets` function.) If you find such a conflict, either:

- Change the common block to be dynamic (using the DC compiler option), or
- Change the name of the common block so it does not conflict with the other language's library routine name.

Resolving library module name conflicts between Fortran and C

Several Fortran and C library routines, shown in Table 4 on page 15, have identical names. To correctly run applications that reference one or more of these names, you need to determine if a name conflict exists, and if so, to resolve it according to the prescription given in this section. Otherwise, a library routine other than the one you intend is likely to be linked into your executable program, and the results during execution will not be what you expect.

Before you proceed with this topic, first resolve any static common block name conflicts as discussed in “Resolving static common block name conflicts” on page 15.

### Table 4: Fortran and C library routine names that are identical

<table>
<thead>
<tr>
<th>Routine names</th>
<th>ABS</th>
<th>ACOS</th>
<th>ASIN</th>
<th>ATAN</th>
<th>ATAN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK</td>
<td>COS</td>
<td>COSH</td>
<td>ERF</td>
<td>ERFC</td>
<td></td>
</tr>
<tr>
<td>EXIT</td>
<td>EXP</td>
<td>GAMMA</td>
<td>LOG</td>
<td>LOG10</td>
<td></td>
</tr>
<tr>
<td>SIN</td>
<td>SINH</td>
<td>SQRT</td>
<td>TAN</td>
<td>TANH</td>
<td></td>
</tr>
</tbody>
</table>

Conditions under which your application does not have a name conflict

If all three of the following conditions are true, your application does not have the name conflict discussed here, and you can therefore skip this topic:

- The *Language Environment interface validation exit* is available. (The interface validation exit is a routine that, when used with the binder, automatically resolves conflicting library routine references within Fortran routines)
  - Under batch, this means any link-edit steps (for example, in cataloged procedures) have been changed to include `EXITS(INTFVAL(CEEPINTV))` in the PARM parameter and to include the following DD statement:

```c
//STEPLIB DD DSN=CEE.SCEELKED,DISP=SHR
```
– Under TSO/E, this means you have included EXITS(INTFVAL(CEEPINTV)) among your link-edit options. Use the TSO/E command TSOLIB to dynamically allocate a STEPLIB.

• You are not relink-editing a pre-Language Environment executable program in which none of the component parts have been changed, but instead are link-editing one or more individual routines.
• None of the routines you are link-editing is an assembler CSECT that references a Fortran library routine from the list in Table 4 on page 15.

Unless all three of the preceding conditions are true, you need to continue reading this section to be able to properly link-edit and run your application.

Determining if your application has a name conflict

Examine Table 5 on page 16. If your application contains a routine that is compiled (assembled) with one of the products shown in column one, and the routine uses one of the functions shown in column two, it has a name conflict that must be resolved.

<table>
<thead>
<tr>
<th>Product used for compilation</th>
<th>Names causing conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS FORTRAN Version 2 Release 5–6</td>
<td>CLOCK, EXIT</td>
</tr>
<tr>
<td>VS FORTRAN Version 2 Release 1–4, or VS FORTRAN Version 1</td>
<td>CLOCK, EXIT, or any name in Table 4 on page 15, if passed as an argument</td>
</tr>
<tr>
<td>FORTRAN IV H Extended, or FORTRAN IV G1, or Assembler, any version</td>
<td>Any name in Table 4 on page 15</td>
</tr>
</tbody>
</table>

To determine how to resolve any name conflicts, determine which of the following conditions (labeled A through E) are true.

A
The Language Environment interface validation exit is available, as described in “Conditions under which your application does not have a name conflict” on page 15.

B
You have a fully executable program created with one or more pre-Language Environment products, and you are not modifying any of its component parts.

C
Condition B is not true, and your application contains at least one assembler CSECT that references a conflicting name listed in Table 4 on page 15. You want the conflicting names in the CSECTs resolved to Fortran routines.

D
Condition B is not true, and your application consists only of one or more individual Fortran or assembler routines, of which at least one references a conflicting name. You want any conflicting names resolved to Fortran routines.

E
Condition B is not true, and your application consists of one or more individual routines that are not just Fortran, assembler, or both. At least one Fortran or assembler routine references a conflicting name, and you want its conflicting names resolved to Fortran routines.

Next, find the row in Table 6 on page 17 that corresponds to the combination of conditions that is true for your application (true conditions are denoted by X, and "don't-care" conditions by –).
Table 6: Decision table for name conflict resolution

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>Do the following:</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>–</td>
<td>–</td>
<td>Do the following: Proceed to “Removing Fortran conflicting references” on page 17.</td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>X</td>
<td>–</td>
<td>–</td>
<td>Proceed to “Relink-editing a pre-Language Environment executable program” on page 20.</td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>X</td>
<td>–</td>
<td>–</td>
<td>Use one of the AFHW* cataloged procedures discussed in “IBM-supplied cataloged procedures” on page 88.</td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>X</td>
<td>–</td>
<td>–</td>
<td>Proceed to “Removing Fortran conflicting references” on page 17.</td>
<td></td>
</tr>
</tbody>
</table>

Removing Fortran conflicting references

For each object or executable program that contains conflicting references that you want resolved to Fortran routines, you must replace the conflicting names with names that are unambiguous, as shown in the examples in this section. Under MVS, you will use the cataloged procedure AFHWN, or under TSO/E, a CLIST, in conjunction with data set SCEESAMP(AFHWNCH) to effect the name replacement. (For information about cataloged procedure AFHWN, see “AFHWN — Resolving name conflicts between C and Fortran” on page 96.)

You can change one or several modules per step, as you wish. Use one of the following examples, adapting it to your application, as needed.

SCEESAMP(AFHWNCH) must be included immediately preceding each individual executable program whose names are to be changed, as shown in each example. AFHWNCH is a data set containing linkage-editor CHANGE statements to change all conflicting names in the module to which it is applied to names known unambiguously as Fortran routines. For example, CHANGE ABS(A#BS) replaces any reference to ABS, a conflicting name, with a reference to A#BS, the Fortran absolute value routine. A complete list of the conflicting names and their corresponding unambiguous Fortran names can be seen in Table 63 on page 392.

The modules resulting from this process have had all their conflicting names replaced. Having no name conflicts, they can, at any time, be linked as part of one or more executable programs in an application, for example:

- Under MVS, by using one of the CEEW* cataloged procedures discussed in “IBM-supplied cataloged procedures” on page 88
- Under TSO/E, by using the LINK command as discussed in “Link-editing your application using the LINK command” on page 72

When a module has had its conflicting references to CLOCK or EXIT changed, it is no longer usable with the VS FORTRAN Version 2 library.

Changing one module per step (MVS)

Under MVS, the example in Figure 5 on page 18 produces one executable program. Conflicting names in USER.INPUT.LOAD(MEM1) are replaced; USER.RESULT.LOAD(MEM1CHG) is the resulting executable program.
Figure 5: Changing conflicting names in an executable program under MVS

The example in Figure 6 on page 18 produces several executable programs. Conflicting names in USER.INPUT.LOAD(MEM1, MEM2, and MEM3) are replaced; the resulting executable programs are USER.RESULT.LOAD(MEM1CHG, MEM2CHG, and MEM3CHG).

Figure 6: Changing conflicting names in several executable programs under MVS

Changing one module per step (TSO/E)

Under TSO/E, the example in Figure 7 on page 18 produces a single executable program. Conflicting names in USER.INPUT.LOAD(MEM1) are replaced; the resulting executable program is USER.RESULT.LOAD(MEM1CHG).

Figure 7: Changing conflicting names in an executable program under TSO/E

Figure 8 on page 19 shows an example that produces several executable programs. Conflicting names in USER.INPUT.LOAD(MEM1, MEM2, and MEM3) are replaced; the resulting executable programs are USER.RESULT.LOAD(MEM1CHG, MEM2CHG, and MEM3CHG).
Figure 8: Changing conflicting names in several executable programs under TSO/E

Changing multiple modules per step (MVS)

Figure 9 on page 19 shows an example for MVS. Conflicting names in USER.INPUT.LOAD(MEM1, MEM2, and MEM3) are replaced; USER.RESULT.LOAD(MEM1CHG) is the resulting executable program. You must explicitly include AFHWNCH before each individual program.

Figure 9: Changing conflicting names in multiple executable programs under MVS

Changing multiple modules per step (TSO/E)

Figure 10 on page 19 contains an example for TSO/E. Conflicting names in USER.INPUT.LOAD(MEM1, MEM2, and MEM3) are replaced; the resulting executable program is USER.RESULT.LOAD(MEM1CHG). You must explicitly link-edit AFHWNCH before each individual executable program.

Figure 10: Changing conflicting names in multiple executable programs under TSO/E
Relink-editing a pre-Language Environment executable program

The action to take to relink-edit a pre-Language Environment executable program depends on whether it contains a reference to one or more of the conflicting names shown in Table 4 on page 15:

• If the executable program contains no reference to any of the conflicting names, but contains parts that reference Fortran routines not in the list of conflicting names, replace the Fortran routines with the equivalent Language Environment routines by using the module as input to the Fortran library module replacement tool, SCEESAMP(AFHWRLK), which is discussed in “Replacing Fortran runtime library modules in a Fortran executable program” on page 14.

• If the executable program does contain parts that reference one or more of the conflicting names, and the names are to be resolved to Fortran routines, the action you take depends on whether C parts are present in the executable program:

  – If the executable program does not contain any C part that references a conflicting name, replace the Fortran routines with the equivalent Language Environment routines by using the executable program as input to the Fortran library module replacement tool, SCEESAMP(AFHWRLK), and link-edit:

    - Under MVS, by using the AFHWL cataloged procedure (see “AFHWL — Link a program written in Fortran” on page 95), as shown in Figure 11 on page 20.

      ```
      //REPFORT    EXEC PROC=AFHWL,PGMLIB=USER.FORT.LOAD
      //USERINP    DD   DSN=USER.FORT.LOAD,DISP=SHR
      //LKED.SYSIN DD   *
      INCLUDE SCEESAMP(AFHWRLK)
      INCLUDE USERINP(MEM1)
      NAME MEM1(R)
      /*
      Figure 11: Replacing Fortran routines with Language Environment routines under MVS
      
      Fortran routines are replaced with the equivalent Language Environment routines using the Fortran library module replacement tool, AFHWRLK. The existing and resulting executable program is USER.FORT.LOAD(MEM1). No DD statement is needed for the SCEESAMP library because it is already included in the AFHWL cataloged procedure.
      
      - Under TSO/E, by using a CLIST as shown in Figure 12 on page 20, Fortran routines are replaced with the equivalent Language Environment routines using the Fortran library module replacement tool, AFHWRLK. The existing and resulting executable program is USER.FORT.LOAD(MEM1).

      PROC 0
      CONTROL MSG NOFLUSH NOPROMPT SYMLIST CONLIST
      LINK ('CEE.SCEESAMP(AFHWRLK)', +
      'USER.FORT.LOAD(MEM1)') +
      LOAD ('USER.FORT.LOAD(MEM1)') +
      LIB ('CEE.SAFHFORT', +
      'CEE.SCEELKED') NOTERM
      /*
      Figure 12: Replacing Fortran routines with Language Environment routines under TSO/E
      
      - If the executable program contains at least one C part that references a conflicting name, you can take one of two possible courses of action depending on whether the individual object modules of the executable program are available to you:

        - If the individual object modules are available, relink-edit the whole application following the name conflict procedure from the beginning to check for possible conflicts, or

        - If the individual object modules are not available, link-edit the executable program:

          • In MVS, using the CEEWL cataloged procedure, as shown in Figure 13 on page 21
          • In TSO/E, using a CLIST, as shown in Figure 14 on page 21

      Do the following in the link-edit step, as shown in the following examples:
1. Include the SAFHFORT library Fortran routines to which the conflicting names should resolve.
2. Include the Fortran library module replacement tool, SCEESAMP(AFHWRLK).
3. Do the CSECT replacement necessary to make the C parts of the executable program compatible. There could be CSECTs that you need to replace in addition to those shown in Figure 13 on page 21 or Figure 14 on page 21; see z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer for details.

The following example relink-edits an executable program containing both C and Fortran (or assembler) routines, where C references SQRT, and Fortran references SIN, LOG, and CLOCK. (The language of the main program here is C. If it were Fortran, the ENTRY CEESTART statement would be rewritten to instead name the Fortran main program.) The existing and resulting executable program is USER.FORTC.LOAD(MEM1).

```
//FORTC      EXEC PROC=CEEWL,PGMLIB=USER.FORTC.LOAD
//USERINP    DD DSNAME=USER.FORTC.LOAD,DISP=OLD
//SAFHFORT   DD DSNAME=CEE.SAFHFORT,DISP=SHR
//SCEESAMP   DD DSNAME=CEE.SCEESAMP,DISP=SHR
//LKED.SYSIN DD *
INCLUDE SAFHFORT(SIN)
INCLUDE SAFHFORT(LOG)
INCLUDE SAFHFORT(CLOCK)
INCLUDE SCEESAMP(AFHWRLK)
INCLUDE USERINP(MEM1)
NAME MEM1(R)
INCLUDE SYSLIB(EDCSTART)
INCLUDE SYSLIB(CEEROOTB)
INCLUDE SYSLIB(@@FTOC)
INCLUDE SYSLIB(@@CTOF)
INCLUDE USERINP(MEM1)
ENTRY CEESTART
NAME MEM1(R)
/*
```

Figure 13: Relink-editing an executable program to resolve conflicting names under batch

The following example relink-edits an executable program containing both C and Fortran (or assembler) routines, where C references SQRT, and Fortran references SIN, LOG, and CLOCK. The existing and resulting executable program is USER.FORTC.LOAD(MEM1).

```
PROC 0
CONTROL MSG NOFLUSH NOPROMPT SYMLIST CONLIST
LINK ('CEE.SAFHFORT(SIN)', +
     'CEE.SAFHFORT(LOG)', +
     'CEE.SAFHFORT(CLOCK)', +
     'CEE.SCEESAMP(AFHWRLK)', +
     'USER.FORTC.LOAD(MEM1)'), +
LOAD ('USER.FORTC.LOAD(MEM1)'), +
LIB ('CEE.SCEELKD') NOTERM
LINK ('CEE.SCEELKD(EDCSTART)', +
     'CEE.SCEELKD(CEEROOTB)', +
     'CEE.SCEELKD(@@FTOC)', +
     'CEE.SCEELKD(@@CTOF)', +
     'USER.FORTC.LOAD(MEM1)'), +
LOAD ('USER.FORTC.LOAD(MEM1)'), +
LIB ('CEE.SCEELKD') NOTERM
```

Figure 14: Relink-editing an executable program to resolve conflicting names under TSO/E
PL/I considerations

This section discusses what you need to know if you link-edit or relink-edit in PL/I.

Link-editing PL/I subroutines for later use

To prelink PL/I subroutines, store them in a load library, and later INCLUDE them with main procedures. The subroutines must be linked with the NCAL link-edit option which causes unresolved external reference error messages from the link-edit process, but these are resolved when the PL/I main procedure is linked with the subroutines. The NCAL option is needed because, in a PL/I load module, all the resident modules must be at the same level. This consistency is ensured because external references are not resolved until the final link.

Replacing PL/I library routines in an OS PL/I executable program

Two jobs, IBMWRLK for batch and IBMWRLKC for CICS, located in the sample library SCEESAMP, replace OS PL/I library routines in an OS PL/I executable program with Language Environment routines. For more information about using IBMWRLK or IBMWRLKC, see IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

Link-editing fetchable executable programs

The PL/I FETCH and RELEASE statements dynamically load separate executable programs that can be subsequently invoked from the PL/I routine that fetches the executable program. There are some restrictions on the PL/I for MVS & VM statements that can be used in fetched procedures. These are described in the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

Many of those restrictions have been removed with Enterprise PL/I for z/OS. See the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735) for the use of FETCH with that compiler.

Fetchable (or dynamically loaded) modules should be link-edited into an executable program library that is subsequently made available for the job step by means of a JOBLIB or STEPLIB DD statement. The FETCH statement can access modules stored in link-pack areas (both the LPA and the ELPA). The search order for modules is defined by z/OS; see “Program library definition and search order” on page 69 for details.

The step that link-edits a fetchable executable program into a library requires the following linkage editor control statements:

- An ENTRY statement to define the entry point into the PL/I routine.
- A NAME statement to define the name used for the fetchable executable program. This statement is required if the NAME compiler option is not used and if the name is not specified in the DSN parameter in the SYSLMOD DD statement used to define the executable program library.

The name or any alias by which the fetchable executable program is identified in the executable program library must appear in a FETCH or RELEASE statement within the scope of the invoking procedure.
Language Environment-conforming COBOL or C modules can be loaded dynamically by the PL/I FETCH statement. The cataloged procedure IEL1CL includes both the compilation and the link-editing of the fetchable PL/I module. For more details on cataloged procedure IEL1CL, see the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

**PL/I link-time considerations**

The product structure for PL/I has changed from the previous PL/I version. Most JCL and CLISTs that link-edit a PL/I application using the OS PL/I library must be changed. These changes include:

- The OS PL/I multitasking library PLITASK has been replaced by SIBMTASK, which is required to have multitasking support. SIBMTASK must be concatenated before SCEELKED.

  Enterprise PL/I for z/OS does not support multitasking. Language Environment continues to support PL/I multitasking for PL/I for MVS & VM as well as previous, supported levels of the PL/I product.

- The PLIBASE and SIBMBASE libraries have been replaced by:
  - SCEELKED, which contains resident routines that are linked with the application and are used to resolve external references at link-edit time.
  - SIBMMATH, which contains the stubs for old OS PL/I V2R3 math library routines. In link-edit steps, this library must precede SCEELKED if old math results are desired in a particular executable program.
  - SIBMSCALL, which is required to provide PLICALLA and PLICALLB compatibility if PL/I for MVS & VM applications use OS PL/I PLICALLA or PLICALLB as an entry point. SIBMSCALL must be concatenated before SCEELKED.
  - SIBMICAL2, which is very similar to SIBMSCALL, but is only used with Enterprise PL/I for z/OS.

  **Note:** SCEELKED and SIBMICAL2 are the only libraries that apply for Enterprise PL/I for z/OS.

**Fetching modules with different AMODEs**

Language Environment supports the PL/I FETCH/RELEASE facility. No special considerations apply to this support when both the fetching executable program and the fetched executable programs have the AMODE(ANY) attribute or both have the AMODE(24) attribute.

Language Environment also supports the fetching of a load module that has a different AMODE attribute than the executable program issuing the FETCH statement. Language Environment performs the AMODE switches in this case, and the following constraints apply:

- If any fetched module is to execute in 24-bit addressing mode, the fetching module must be loaded into storage below 16M, and must have the RMODE(24) attribute regardless of its AMODE attribute.
• Any variables passed as parameters to a fetched routine must be addressable in the AMODE of the fetched procedure. For any fetched executable program that is to be executed in 24-bit addressing mode, you must ensure that:
  – If any parameter resides in a HEAP area, the BELOW suboption of the HEAP option is specified.
  – If any parameter resides in STATIC storage of the fetching executable program, the fetching executable program has the RMODE(24) attribute so that its STATIC storage is below 16M.
  – If any parameter resides in AUTOMATIC storage, no special considerations apply because ALL31(OFF) and STACK(,,BELOW) runtime options have been used. If the two constraints described previously cause problems, then you can copy the variable to a like variable with the AUTOMATIC attribute and pass the copy to the fetched AMODE(24) procedure, with the BELOW suboption of the HEAP option specified.

• PL/I object modules can be link-edited into overlay executable programs and run as overlay executable programs. Such programs have the attributes AMODE(24) and RMODE(24).

When a PL/I routine fetches another PL/I procedure, it is possible for a condition to arise in the fetched procedure for which a PL/I ON-unit was established in the fetching procedure.

PL/I imposes the restriction that if an ON-unit is established while the current addressing mode is 24-bit, and the condition is raised while the addressing mode is 31-bit, the ON-unit is not invoked. This is because PL/I must invoke the ON-unit in the addressing mode in which it was established. If the ON-unit was established in 24-bit addressing mode but the condition arose in 31-bit addressing mode, the code and data required to process the error might not even be addressable in 24-bit addressing mode.
Chapter 3. Using Extra Performance Linkage (XPLINK)

What is XPLINK?

Extra Performance Linkage (XPLINK) is a call linkage between programs that has the potential for a significant performance increase when used in an environment of frequent calls between small functions or subprograms.

Objectives

The C/C++ subroutine linkage on z/OS cannot be considered state-of-the-art with respect to performance. It represents a disproportionate percentage of total execution time, higher yet for C++ than for C due to the many, typically small, functions. Depending on the style of programming, the total prolog/epilog cost may reach a double digit percentage even for C, and thus represents a significant potential for further program optimization.

The objective of XPLINK is to significantly speed up the linkage for C and C++ routines by using a downward-growing stack and by passing parameters in registers. It includes support for reentrant and non-reentrant code, for calls to functions in DLLs, and compatibility with old code.

With XPLINK, the linkage and parameter passing mechanisms for C and C++ are identical. If you link to a C function from a C++ program, you should still specify `extern C` to avoid name mangling.

The primary objective of XPLINK is to make subroutine calls as fast and efficient as possible by removing all nonessential instructions from the main path.

This is achieved by introducing the following:

- Stack growth from higher to lower addresses ("negative-" or "downward-growing"):
  - To eliminate overhead in stack frame allocation
  - To eliminate the need to check for inline stack overflow
  - To allow an improved epilog
  - To allow addressability to information (such as parameters) in the caller's stack frame
- Biasing the stack pointer (by 2048 bytes), so that small functions can save registers in their own stack frame before updating the stack pointer, avoiding address generation interlocks.
- Reassignment of registers (see “XPLINK register conventions” on page 30) to support more efficient saving and restoring of registers in function prologs and epilogs.
- Parameter passing in registers and accepting return values in registers.
- Elimination of Interlanguage Call (ILC) overhead (marking of stack frame) for non-ILC calls.
- Faster call sequences for inter-module calls.
- Passing the address of the data area associated with a function, its "environment," to the function on entry.
- No branching around Language Environment words.
- Use of relative branching for function calls where possible.
- Unification of the various (RENT and NORENT, DLL, and NODLL) function pointer implementations, reducing the costs of all operations involving function pointers.

An important additional objective is reducing the module size in memory, which is accomplished by eliminating unused information in function blocks.
Support for XPLINK

XPLINK support is available for applications running under the following environments:

- Batch
- TSO/E
- z/OS UNIX

It is not available for applications running under CICS before CICS TS 3.1.

XPLINK support is available with the compiler for the following programming language:

- z/OS C and C++

There is limited XPLINK support in the following areas:

- Db2 — EXEC SQL calls are defined using linkage OS which is supported from XPLINK callers.
- IMS — Language Environment provides the CTDLI interface (a ctdli() function call) for C and C++ callers. This interface is defined in the ims.h header as using linkage OS which is supported from XPLINK callers.
- In general, any system service that is defined as using linkage OS is a supported call from an XPLINK program.
  - If it requires OS linkage conventions but not a Language Environment-conforming stack (that is, it only needs a 72-byte save area), then the function can be defined as OS_NOSTACK (the default when #pragma linkage(...) is specified). This option provides the best performance because the compiler generates OS linkage calling conventions directly – no call through glue code is required.
  - If it requires OS linkage conventions and a Language Environment-conforming stack, then the function can be defined as OS_UPSTACK. For this option, the compiler generates a call through Language Environment glue code that switches to OS linkage conventions and the non-XPLINK upward-growing stack.

For more information, see either z/OS Language Environment Writing Interlanguage Communication Applications or z/OS XL C/C++ Language Reference.

For more information, see “XPLINK restrictions” on page 35.

XPLINK concepts and terms

**XPLINK**

Extra Performance Linkage (XPLINK) is a new call linkage between programs which has the potential for a significant performance increase when used in an environment of frequent calls between small functions or subprograms.

**non-XPLINK application**

A non-XPLINK application is one in which none of the executables involved have been compiled with the XPLINK compiler option specified.

**XPLINK application**

An XPLINK application is one in which at least one of the executables involved as been compiled with the XPLINK compiler option specified. XPLINK and non-XPLINK compiled source code cannot be link-edited together into the same executable, but XPLINK and non-XPLINK executables (for example, DLLs) can be mixed in the same application. The performance advantage from XPLINK is increased as the percentage of XPLINK executables in an application increases.

**XPLINK environment**

An XPLINK environment is one in which Language Environment has initialized the necessary resources to run an XPLINK application (for example, a downward-growing stack). This is accomplished by either invoking an initial program that was compiled with the XPLINK compiler option specified, or specifying the XPLINK(ON) runtime option.

**downward-growing stack**

The standard Language Environment stack is upward-growing. For XPLINK, a main feature of its more efficient program prolog code is a program stack which grows from higher to lower addresses. This
provides implicit protection against exceeding available stack storage, rather than having to make an explicit test, and therefore reduces path length.

**guard page**

A write-protected area of storage at the low address end of a downward-growing stack segment. This allows a stack frame (smaller than the size of the guard page) to be allocated by storing into the low address of the stack frame. Stack segment overflow and extension is triggered by the exception resulting from a prolog storing into the guard page (implicit stack overflow detection).

**glue code**

With respect to XPLINK compatibility, glue code refers to the code inserted between XPLINK and non-XPLINK executables, which converts the stack structure, registers and parameter list into a format suitable for the called function, and then restores the environment upon return.

**The XPLINK stack**

Stack storage is automatically created by Language Environment and is used for routine linkage and automatic storage. This topic describes the way the XPLINK stack differs from the standard Language Environment stack, which is described in detail in “Stack storage overview” on page 148.

The prolog of a function usually allocates space (referred to as a “frame”, “Stack Frame", or "DSA" - dynamic storage area) in the Language Environment-provided stack segment for its own purposes and to support calls to other routines.

Figure 16 on page 27 shows the structure of the standard Language Environment stack. Note that the DSAs in the standard (upward-growing) Language Environment stack are allocated from lower to higher addresses. Figure 17 on page 28 shows how the XPLINK (downward-growing) stack is different, specifically that the DSAs are allocated from higher to lower addresses, with the presence of the guard page to mark the bottom of the stack.
The XPLINK stack frame layout

Figure 18 on page 29 shows the XPLINK stack frame layout.

The XPLINK stack register (general-purpose register (GPR) 4) is biased, meaning it points to a location 2048 bytes before the stack frame for the currently active routine. It grows from numerically higher storage addresses to numerically lower ones, that is, the stack frame for a called function is normally at a lower address than the calling function. The stack frame is aligned on a 32-byte boundary.
Save area (48 bytes)

This area is always present when a stack frame is required. It holds up to 12 registers. The first two words hold, optionally, GPRs 4 and 5, which contain the address of the previous stack frame and the environment address passed into the function. This is followed by the two words containing GPR 6, which may or may not hold the actual entry point address (depending on the type of call), and GPR 7, the return address. GPR 8 through GPR 15, as used by the called function, are saved in the following 32 bytes. Table 7 on page 31 shows the XPLINK register conventions.

Except when registers are saved in the prolog, this area may not be altered by compiled code. The PPA1 GPR Save Mask indicates which GPRs are saved in this area by the prolog.

Stack overflow is detected when the STM instruction stores into the write-protected guard page while trying to save the registers in this save area.

Storage of the Backchain field in the save area is triggered by the optional XPLINK(BACKCHAIN) compiler option (or at the convenience of the compiler). This adds slightly to the cost of the prolog code, but may increase the serviceability characteristics of the application by providing a DSA backchain pointer in the save area. Note that this backchain pointer may or may not be valid, depending on the options specified when a function was compiled.

The third slot in the save area contains the value in GPR 6 on entry to the routine. If the routine was called with a BASR instruction, this will be the address of the function's entry point. The fourth slot contains the return address. The return point can be examined to determine how the function was called: if with a BASR instruction then the entry point address can be found in the third slot of the save area; if with a branch relative instructive, then the entry point can be computed from the return address and the branch offset contained in the branch relative instruction.
**Reserved (8 bytes)**
This area is always present and is for the exclusive use of the runtime environment. It is uninitialized by compiled code.

**Argument area prefix (4 bytes)**
This area is always present and is for the exclusive use of the runtime environment. It is uninitialized by compiled code.

**Argument area (minimum 16 bytes)**
This area (at a fixed DSA offset of 64 bytes into the caller’s stack frame) contains the argument lists which are passed on function calls made by the function associated with this stack frame. The called function finds its parameters in the caller’s stack frame. Arguments passed in registers are not present in the argument area in the save area. This can be overridden by the optional XPLINK(STOREARGS) compiler option. This adds slightly to the cost of the prolog code, but might increase the serviceability characteristics of the application by providing a complete record of the arguments passed as input to a function.

A minimum of four words (16 bytes) is always allocated.

**Local storage**
Local storage is the space owned by the executing procedure. It can be used for its local variables and temporaries.

**Total stack frame size**
The total stack frame size is calculated by adding the preceding fields and by rounding the sum up to a quadword boundary.

**Stack overflow**
To maximize function call performance, XPLINK replaces the explicit inline check for overflow with a storage protect mechanism that detects stores past the end of the stack segment.

The stack floor is the lowest usable address of the current stack segment. Toward lower addresses, it is preceded by a store-protected guard page used to detect stack overflows.

Availability of space for a stack frame is ensured in the function prolog usually by storing into the start of the called function's frame. In case of overflow, this triggers an exception which in turn causes a discontinuous extension of the stack by Language Environment. Functions with a DSA larger than the guard page use the stack floor address in the CAA to verify space availability. Allocation and deallocation of extensions is transparent to the application.

To make the stack appear contiguous to the application, a small stack frame containing all fields up to and including the Argument area will be allocated in the new stack segment for use by the called function, and the contents of the caller’s stack, up to the end of the argument area, is copied into the new stack segment. The length of the argument list expected is available in the called function's PPA1 except for vaar args functions, where the entire argument area in the calling function must be copied.

Stores into the guard page done outside the prolog done outside alloca() built-in processing should be treated as invalid and cause the application to be terminated.

Stack overflow is detected by the STM instruction that is used to save registers in this save area.

Initial stack segment size and the stack segment increment size are controlled by the STACK runtime option (see z/OS Language Environment Programming Reference).

**XPLINK register conventions**
XPLINK introduces a register scheme which is different from standard OS linkage, in order to optimize the performance of saving and restoring registers in function prologs and epilogis. The registers which are saved in the register save area of the XPLINK stack frame are described in “Save area (48 bytes)” on page 29.
Table 7 on page 31 shows the layout of the XPLINK interface.

<table>
<thead>
<tr>
<th>Interface part</th>
<th>Non-XPLINK</th>
<th>XPLINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack pointer</td>
<td>Reg 13</td>
<td>Reg 4 (biased; see “The XPLINK stack frame layout” on page 28)</td>
</tr>
<tr>
<td>Return address</td>
<td>Reg 14</td>
<td>Reg 7</td>
</tr>
<tr>
<td>Entry point on entry</td>
<td>Reg 15</td>
<td>Reg 6 (not guaranteed; a routine may be called via branch relative)</td>
</tr>
<tr>
<td>Environment</td>
<td>Reg 0 (writable static)</td>
<td>Reg 5</td>
</tr>
<tr>
<td>CAA Address</td>
<td>Reg 12</td>
<td>Reg 12</td>
</tr>
<tr>
<td>Input parameter list</td>
<td>address in R1</td>
<td>Located at offset 2112 ('840'x) off R4 (fixed location in caller's stack frame). First three words are passed in R1-R3, floating point values in FRP0, 2, 4, 6.</td>
</tr>
<tr>
<td>Return code</td>
<td>Reg 15</td>
<td>R3 (extended return value in R1,R2)</td>
</tr>
<tr>
<td>Start address of callee's stack frame</td>
<td>Caller's NAB value</td>
<td>Caller's Reg 4 - DSA size</td>
</tr>
<tr>
<td>End address of callee's stack frame</td>
<td>Caller's NAB value + DSA size</td>
<td>Caller's Reg 4</td>
</tr>
<tr>
<td>Where caller's registers are saved</td>
<td>R0-R12 saved in caller's stack frame R13 saved in callee's stack frame R14-R15 saved in caller's stack frame</td>
<td>R0 not saved, not preserved R1-R3 not saved, not preserved R4 not saved, recalculated (or saved, restored) R5 not saved, not preserved R6 saved in callee's stack frame, not restored R7-R15 saved in callee's stack frame (R7 is the return register and is not guaranteed to be restored)</td>
</tr>
</tbody>
</table>

See z/OS Language Environment Vendor Interfaces for additional information about register usage and conventions.

**XPLINK parameter passing and return code conventions**

XPLINK uses a logical argument list consisting of contiguous 32-bit words, where some arguments are passed in registers and some in storage.

The argument list is located in the caller's stack frame at a fixed offset (+2112) from the stack register (GPR 4). It provides space for all arguments, including those passed in registers. It also includes an extra unused word (4 bytes), which might be required in compatibility situations, at the end of the argument area. Its size is sufficient to contain all the arguments passed on any call statement from a procedure associated with the stack frame.

Function return values are also returned in registers. When the return value will not fit in registers, it is always returned in a buffer allocated by the caller. For more details on XPLINK parameter and return value conventions, see z/OS Language Environment Vendor Interfaces.
When XPLINK should be used

The type of application that could benefit most from using XPLINK is an application that makes many calls to small functions. C++ is a good example, since the OO programming model makes this possible. C applications that make many function calls might also be suitable for XPLINK.

To further enhance the performance of an XPLINK application, the IEEE binary floating-point math library should be used by specifying the FLOAT(IEEE) compiler option. This math library has been recompiled entirely in XPLINK, while the Hexadecimal math library remains non-XPLINK and therefore requires a call through glue code from XPLINK applications.

When XPLINK should not be used

Functions compiled XPLINK and NOXPLINK cannot be combined in the same program object (except when the #pragma linkage(OS) directive is used in C, or extern in C++; see z/OS Language Environment Writing Interlanguage Communication Applications for more information).

While XPLINK can provide a significant performance enhancement to the right application, it can also degrade the performance of an application that is not suitable for XPLINK.

One way to call an XPLINK function from non-XPLINK is to use the DLL call mechanism. But there is an overhead cost associated with calls made from non-XPLINK to XPLINK, and from XPLINK to non-XPLINK. This overhead includes the need to swap from one stack type to another and to convert the passed parameters to the style accepted by the callee. Applications that make many "cross-linkage" calls might lose any benefit obtained from the parts that were compiled XPLINK, and in fact performance could be degraded from the pure non-XPLINK case. If the number of pure XPLINK function calls is significantly greater than the number of "cross-linkage" calls, then the cost saved on XPLINK calls will recover some of the costs associated with calls that involve stack swapping.

When you introduce an XPLINK program object into your application, for example an XPLINK version of a vendor-DLL which your application uses, then your application must now run in an XPLINK environment (this is controlled by the XPLINK runtime option). In an XPLINK environment, an XPLINK version of the C/C++ runtime library (RTL) is used. You cannot have both the non-XPLINK and XPLINK versions of the C/C++ RTL active at the same time, so in an XPLINK environment, non-XPLINK callers of the C/C++ runtime library also incurs this stack swapping overhead.

The maximum performance improvement can be achieved by recompiling an entire application XPLINK. The further the application gets from pure XPLINK, the less the performance improvement, and at some point you might see a performance degradation.

The only compiler that supports the XPLINK compiler option is the z/OS XL C/C++ compiler. All COBOL and PL/I programs are non-XPLINK, and therefore calls between COBOL or PL/I and XPLINK-compiled C/C++ are cross-linkage calls and will incur the stack swapping overhead. For more information about making ILC calls with XPLINK, see z/OS Language Environment Writing Interlanguage Communication Applications.

If the application contains C or C++ and the XPLINK(ON) runtime option is specified, then the XPLINK-compiled version of the C runtime library (RTL) is loaded, which will run on the downward-growing stack. When non-XPLINK functions call C RTL functions in this environment, a swap from the upward-growing stack to the downward-growing stack will occur. This results in additional overhead that could cause a performance penalty. Applications that make heavy use of the C RTL from non-XPLINK callers should be aware of this, and if necessary for performance reasons, either run in a pure non-XPLINK environment with XPLINK(OFF) (the default in this case), or convert as much of the application to XPLINK as possible and run with XPLINK(ON).

Applications that use Language Environment environments that are not supported in an XPLINK environment, or that use products that are not supported in an XPLINK environment (for example, CICS before CICS TS 3.1), cannot be recompiled as XPLINK applications.
How is XPLINK enabled?

XPLINK is enabled on several levels, including a compiler option and several runtime options.

XPLINK compiler option

The z/OS XL C/C++ XPLINK compiler option produces an object that uses the XPLINK calling conventions. This compiler option is described in detail in z/OS XL C/C++ User’s Guide.

XPLINK runtime option

Language Environment initializes the enclave as an XPLINK environment if the initial program is compiled XPLINK or the XPLINK(ON) runtime option is specified. If the initial program is non-XPLINK but may call an XPLINK program later in its execution, then the XPLINK(ON) runtime option is required so that the XPLINK resources will be allocated and available when they are needed.

Applications that consist only of non-XPLINK functions (for example COBOL or PL/I) should not execute with the XPLINK(ON) runtime option, because this option provides no benefit when not running an XPLINK application, and could result in performance degradation. In fact, for non-XPLINK applications, enabling this runtime option could result in abends for applications that have not been tested to run in an XPLINK environment, for example, if they use resources or subsystems that are restricted in an XPLINK environment. See “XPLINK restrictions” on page 35.

No AMODE 24 routines are allowed in an enclave that uses XPLINK. When an application is running in an XPLINK environment (that is, either the XPLINK(ON) runtime option was specified, or the initial program was compiled XPLINK), the ALL31 runtime option will be forced to ON. No message will be issued to indicate this action. In this case, if a Language Environment runtime options report is generated using the RPTOPTS runtime option, the ALL31 option will be reported as "Override" under the LAST WHERE SET column.

When an application is running in an XPLINK environment (that is, either the XPLINK(ON) runtime option was specified, or the initial program was compiled XPLINK), the STACK runtime option will be forced to STACK(,,ANY). Only the third suboption of the STACK runtime option is changed by this action, to indicate that stack storage can be allocated anywhere in storage. No message will be issued to indicate this action. In this case, if a Language Environment runtime options report is generated using the RPTOPTS runtime option, the STACK option will be reported as Override under the LAST WHERE SET column.

Related runtime options

The STACK runtime option controls the allocation of the thread's stack storage for the standard Language Environment upward growing stack and the XPLINK downward-growing stack. STACK controls storage allocation for the initial thread in a multi-threaded application.

Similarly, the THREADSTACK runtime option controls the allocation of stack storage for the upward and downward-growing stacks, for other than the initial thread in a multi-threaded application. The THREADSTACK runtime option replaces the NONIPTSTACK runtime option. The NONIPTSTACK runtime option remains for compatibility, but was not enhanced for XPLINK.

Building and running an XPLINK application

The detailed procedures for building and running non-XPLINK Language Environment-conforming applications can be found in other topics in this information.

The procedures for building XPLINK Language Environment-conforming applications can be summarized as:

1. Compile the application with an XPLINK compiler (the z/OS XL C/C++ compiler) using the XPLINK compiler option.
2. Link edit, with the DFSMS binder, the application (specifying a PDSE or HFS file as the output data set) with the object files and the following Language Environment input:

- Where SYSLIB for non-XPLINK applications usually lists the SCEELKD, SCEELKEX, SCEEOBJ, and SCEECPP data sets, the SYSLIB for link-editing an XPLINK application replaces these with the SCEEBND2 data set. SCEEBND2 contains all object files necessary for building Language Environment-conforming XPLINK applications. If you attempt to link edit an XPLINK application using the non-XPLINK static libraries, or vice versa, you will receive the binder error message IEW2469E indicating a mismatch in linkage type between function reference and definition.

- If the XPLINK application calls C runtime library (RTL) functions, it must include the XPLINK C RTL sidedeck CELHS003 that is in the SCEELIB data set. This is included automatically by c89 when the -Wl, xplink option is specified. See z/OS UNIX System Services Command Reference or z/OS XL C/C++ User’s Guide for information about building XPLINK applications in batch, TSO/E, and z/OS UNIX.

- If the XPLINK application calls Language Environment AWIs or CWIs, it must include the XPLINK Language Environment sidedeck CELHS001 that is in the SCEELIB data set.

- If the XPLINK application is written in C++, it picks up Language Environment C++ RTL definitions from the XPLINK C++ sidedeck CELHSCPP that is in the SCEELIB data set. This sidedeck is used instead of the SCEECPP data set (the SCEECPP data set is used by non-XPLINK applications).

3. Run the application by providing both SCEERUN and SCEERUN2 data sets in the MVS program search order, for example STEPLIB or LNKLST.

SCEERUN and SCEERUN2 can be specified in any search order. The XPLINK(ON) runtime option is required if the initial program in the application is non-XPLINK and XPLINK programs can be called (via DLL).

Other considerations

When you compile and link edit a program, the resulting executable is either XPLINK or non-XPLINK. That is, XPLINK-compiled parts and NOXPLINK-compiled parts cannot be link-edited together in the same program object. The one exception to this is the use of the #pragma linkage(OS) directive for C (or extern "OS" for C++). The intent here is to allow the calling of existing assembler programs that typically perform some function that cannot be done in C or C++ without having to rewrite the assembler program using XPLINK conventions (although this could be done if performance is critical, see the topic on combining C or C++ and assembler in z/OS XL C/C++ Programming Guide). XPLINK and non-XPLINK executables can be mixed at run time, for example by using DLL function calls. An XPLINK function can call a non-XPLINK function in a separate DLL, and vice versa. If needed, glue code is inserted automatically by Language Environment to perform the necessary stack switching and parameter passing adjustments.

The existing static/resident libraries cannot be used when building XPLINK applications. They contain static parts that get resolved by the binder before the entries in the XPLINK sidedecks.

The DFSMS binder must be used to create an XPLINK application. The resulting program module exploits the format of the PM3 Program Object.

There are also XPLINK versions of locales and iconv converters that are provided for use by XPLINK applications.

There is an PLINK-compiled version of the Curses archive file. It is called libcursesxp.a, and resides in /usr/lib. The usage is the same as the old archive file except the compiler and environment must be set up using XPLINK. The following is an example of compiling test.c with the Curses XPLINK archive:

```
c89 -o test -Wc,xplink -Wl,xplink test.c -lcursesxp
```

XPLINK / non-XPLINK compatibility

Compatibility with XPLINK only exists for Language Environment-conforming non-XPLINK applications that are able to run AMODE(31).
XPLINK Compatibility Support is defined as the ability for programs compiled NOXPLINK to transparently call programs that are compiled XPLINK, and vice versa. The programs can be non-XPLINK C or C++ (Fastlink), COBOL, PL/I or OS Linkage Assembler.

This transparent compatibility is provided at the Program Object boundary. It is also provided at the load module boundary, for compatibility with prelinker-built executables. That is, a Program Object (or load module) containing a caller of one linkage type (XPLINK or NOXPLINK) can call a function compiled with the opposite linkage type as long as the called function resides in a different Program Object or load module. Program Objects can reside in either a PDSE or the HFS; load modules reside in PDSs.

Compatibility requires that the differences between stack structures, register conventions, and parameter lists are handled. Language Environment will automatically insert the glue code that performs the necessary transitions between XPLINK and non-XPLINK functions.

The main call linkage supporting XPLINK Compatibility is the DLL call mechanism, but C’s `fetch()` and Language Environment’s CEEFETCH Assembler macro are also supported.

The following are not supported for XPLINK:
- COBOL dynamic call of an XPLINK function
- PL/I FETCH
- CEELOAD

**XPLINK restrictions**

- In general, XPLINK-compiled objects cannot be statically bound with non-XPLINK-compiled objects. A program object (or load module) consists of either XPLINK objects or non-XPLINK objects.

  The one exception to this is when an XPLINK function calls a function that is defined as either OS_UPSTACK or OS_NOSTACK. In this case, the called function is non-XPLINK and uses OS linkage conventions. However, since the bind step of XPLINK and non-XPLINK executables uses different data sets (see “Planning to link-edit and run” on page 7), all external references from the non-XPLINK function must be resolved using the XPLINK link-edit data sets. For example, if a called OS_UPSTACK function makes a call to the C runtime (RTL), the C RTL function must be resolved via the CELHS003 sidedeck in SCEELIB. It cannot use the SCEELKED static stubs since these are not used to bind XPLINK objects.

  The intent of OS_UPSTACK is to be able to call a non-XPLINK function that is not going to be recompiled or rewritten as XPLINK, but is itself a leaf routine and does not make any further calls. The intent of OS_NOSTACK is to call non-XPLINK functions that only need an OS linkage register save area, and are either leaf routines or make calls to other system services that do not use the Language Environment stack.

- XPLINK Assembler programs cannot resolve the address and environment of other XPLINK functions in order to call them. If a function pointer is passed to an XPLINK Assembler program, it can be used to call that function as long as XPLINK calling conventions are used (see z/OS Language Environment Vendor Interfaces). There is no CALL macro support for XPLINK.

  The intent of the XPLINK Assembler support is to be able to call an Assembler function that was rewritten using XPLINK conventions (either for performance reasons or to perform some function that is not easily implemented in C or C++), but is itself a leaf routine and does not make any further calls.

- Calls between XPLINK and non-XPLINK functions are allowed when they cross program object (or load module) boundaries. The use of DLLs is the primary method, where a function in a non-XPLINK DLL calls another function in an XPLINK DLL (or vice versa). The `fetch()` function also provides compatibility between XPLINK and non-XPLINK functions.

- The following do not support calls to XPLINK functions:
  - COBOL dynamic call
  - PL/I FETCH
• XPLINK functions can only call non-XPLINK functions that are also Language Environment-conforming, that is, they were compiled using a Language Environment-conforming compiler.
• XPLINK applications must run AMODE 31, so the ALL31 runtime option will be forced ON. This means all non-XPLINK applications that can call or be called by an XPLINK application must also run AMODE 31.
• Make a reference from XPLINK code into non-XPLINK code only if the reference is by an imported function or variable, or the function pointer is a parameter into the XPLINK code. This prevents incompatible references to a non-XPLINK function entry point.
• XPLINK applications must be built using the DFSMS Binder, and they must reside in either a PDSE or the HFS. The Prelinker cannot be used to create an XPLINK application.
• The following environments and subsystems do not support applications that have been compiled XPLINK:
  – Releases of CICS TS before CICS TS 3.1 (Chapter 25, “Running applications under CICS,” on page 351)
  – Procedures stored by Db2 cannot be compiled XPLINK (Chapter 26, “Running applications under Db2,” on page 365)
  – A nested (child) enclave must run with the same XPLINK environment as its parent (Chapter 31, “Using nested enclaves,” on page 469)
  – The CEEBXITA and CEEBINT user exits cannot be XPLINK (Chapter 28, “Using runtime user exits,” on page 373)
  – PICI
  – System Programmer C (SPC)
  – C Multitasking Facility (C MTF)
  – PL/I Multitasking
Chapter 4. Building and using dynamic link libraries (DLLs)

The z/OS dynamic link library (DLL) facility provides a mechanism for packaging programs and data into load modules (DLLs) that may be accessed from other separate load modules. 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 load module as the referencing routine. When an application references a separate DLL for the first time, it is automatically loaded into memory by the system.

There are two types of DLLs: simple and complex. A simple DLL contains only DLL code in which special code sequences are generated by the compiler for referencing functions and external variables, and using function pointers. With these code sequences, a DLL application can reference imported functions and imported variables from a DLL as easily as it can non-imported ones.

A complex DLL contains mixed code, that is, some DLL code and some non-DLL code. A typical complex DLL might contain some C++ code, which is always DLL code, and some C object modules compiled with the NODLL compiler option bound together.

This topic defines DLL concepts and shows how to build simple DLLs and DLL Applications.

Support for DLLs

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

- z/OS batch
- CICS
- IMS
- TSO
- z/OS UNIX

It is not available for applications running under SP C, CSP or MTF.

**Note:** For CICS, all potential DLL executable modules are registered in the CICS PPT control table in the CICS environment and are invoked at run time.

DLL support is available with the compilers for the following programming languages:

- C and C++
- Enterprise COBOL for z/OS
- COBOL for OS/390 & VM
- Enterprise PL/I for z/OS and OS/390
- High Level Assembler (HLASM) Release 5

**Note:** PL/I for MVS & VM and OS PL/I 2.3 do not support the creation of DLLs or the calling of DLLs.

DLL concepts and terms

**Function**

In this topic, function is used to generically refer to a callable routine or program, and is specifically applicable to C and C++. In COBOL a function would be a COBOL program or method. In Enterprise PL/I a function would be a PL/I procedure.
Variable
In this topic, 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 executable program module 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. It is important to note that compiling code with the DLL option does not mean that the produced executable will be a DLL. To create a DLL, you must compile with the DLL option and export one or more symbols.

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

DLL code
DLL code is code that is compiled with the DLL option of the C and COBOL compilers, code that is compiled with the RENT option of the Enterprise PL/I compiler, or any code compiled with the C++ compiler.

Executable program (or executable module)
A file which can be loaded and executed on the computer. z/OS supports two types:

- Load module
  An executable residing in a PDS.

- Program object
  An executable residing in a PDSE or in the z/OS UNIX file system.

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 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.

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

Variable descriptor
An internal control block containing information about the variable needed by compiled code.

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 d11load() 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 and variable descriptors.

If a C++ DLL contains static classes, their constructors are run when the DLL is loaded, typically before the main function runs. 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.

To implicitly load a DLL from COBOL, do one of the following:

• Call a function that is exported from the DLL.
• Set a COBOL procedure-pointer to a function that is exported from the DLL.
•Invoke a method that is defined in a class contained in the DLL.

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 runtime.

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
• dlclose(), which frees a DLL that was loaded with dlopen()
• dl sym(), 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 only be used with dllqueryfn(), dllqueryvar(), or dllfree(). And a handle returned by dlopen() can only be used with dl sym() and dlclose().

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

For more information about the C runtime services, see z/OS XL C/C++ Runtime 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.

• 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 <dlfcn.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.

“Explicit use of a DLL in a C application” on page 40 and “Explicit use of a DLL in a COBOL/C application” on page 42 have 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.

```c
#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
"  <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);
  }
  dll  = argv[1];
  type = argv[2];
  id   = argv[3];
  dllHandle = dllload(dll);
  if (dllHandle == NULL) {
    perror("DLL-Load");
    fprintf(stderr, "Load of DLL %s failed\n", dll);
    return(8);
  }
  if (strcmp(type, FUNCTION)) {
    if (strcmp(type, VARIABLE)) {

```

The following example shows explicit use of a DLL in an application using the dlopen() family of functions.

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

typedef int (DLL_FN)(void);

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\n" " <type>)\n", progName);
    return;
}

int 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, RTLD_LAZY);
    if (dllHandle == NULL) {
        Syntax(argv[0]);
        return(4);
    }
    symPtr = dlsym(dllHandle, id);
    if (symPtr == NULL) {
        Syntax(argv[0]);
        return(4);
    }
    value = (int)symPtr;
    printf("Result of call to %s is %d\n", id, value);
    dlclose(dllHandle);
    return(0);
}
```

Dynamic link libraries (DLLs)

Building and using dynamic link libraries (DLLs)
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);
}

For more information on the DLL functions, see z/OS XL C/C++ Runtime Library Reference.

Explicit use of a DLL in a COBOL/C application

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

CBL NODYNAM
IDENTIFICATION DIVISION.
PROGRAM-ID. 'COBOL1'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING- STORAGE SECTION.
01 DLL-INFO.
   03 DLL-LOADMOD-NAME PIC X(12).
   03 DLL-PROGRAM-NAME PIC X(160).
   03 DLL-PROGRAM-HANDLE POINTER.
77 DLL-RC PIC S9(9) BINARY.
77 DLL-PROGRAM-PTR PROCEDURE-POINTER.
77 DLL-STATUS PIC X(1) VALUE 'N'.
   88 DLL-LOADED VALUE 'Y'.
   88 DLL-NOT-LOADED VALUE 'N'.
PROCEDURE DIVISION.
* IF DLL-NOT-LOADED
   THEN
     * Move the names in. They must be null terminated.
     MOVE Z'00C05R' TO DLL-LOADMOD-NAME
     MOVE Z'00C05R' TO DLL-PROGRAM-NAME
*   Call the C routine to load the DLL and to get the
*   function descriptor address.
   CALL 'A1CCDLGT' USING BY REFERENCE DLL-INFO
Managing the use of DLLs when running DLL applications

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

Loading DLLs

When you load a DLL for the first time, either implicitly or via an explicit d11load() or d10pen(), writable static 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 file system 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
Dynamic link libraries (DLLs)

- If the file has an unambiguous z/OS UNIX name (it starts with a ./ or contains a /), the file is searched for only in the z/OS UNIX file system.
- 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 z/OS UNIX file system. This is done using the system service BPX1LOD. 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 z/OS UNIX file system 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 z/OS UNIX file system is loaded.
  4. If the DLL is found and does not fall into one of the previous two cases, a load from the z/OS UNIX file system is attempted.

If the DLL could not be loaded from the z/OS UNIX file system because the file was not found or the application does not have sufficient authority to search for or read that file (that is, BPX1LOD fails with errnos 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 BPX1LOD, the load of the DLL is terminated. For an explicit DLL load, the error is reported with the errno and errnojr values set. Correct the indicated error and rerun the application.

If the DLL could not be loaded from the z/OS UNIX file system, 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 z/OS UNIX file system.

**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 symbolic links) might result in a decrease in DLL load performance. The use of 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 loads).

Changing the search order for DLLs while the application is running (for example, changing LIBPATH) might 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.

DLLs are not shared in a nested enclave environment. Only the enclave that loaded the DLL can access functions and variables.

Freeing DLLs

You can free explicitly loaded DLLs with a dllfree() or dlclose() 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 a DLL or a DLL application is similar to creating a C, C++, COBOL or Enterprise PL/I application. It involves the following steps:

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

For more information, see z/OS XL C/C++ Programming Guide, the appropriate version of the programming guide in the COBOL library at Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733), or the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

Building a simple DLL

This topic shows how to build a simple DLL. See “Building a simple DLL application” on page 50 for information about building a simple DLL application.

Writing DLL code
Writing your C DLL code

To build a simple C DLL, write code using the `#pragma export` directive to export specific external functions and variables as shown in Figure 19 on page 46.

```c
#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 19: 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 simple 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`:

```c
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 20: Using `#pragma export` to create a DLL executable module TRIANGLE

Dynamic link libraries (DLLs)
• Do not inline the function if you apply the _Export keyword to the function declaration.

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

*Figure 21: 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.

```cpp
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 COBOL DLL code

There are no special DLL conditions for writing your COBOL code.

### Writing your Enterprise PL/I DLL code

Any PL/I routine other than an OPTIONS(MAIN) procedure can go into a DLL. A package containing a MAIN procedure cannot go into a DLL. Only those external variables that have the RESERVED attribute are exported from a package.

### Writing your Language Environment-conforming assembler DLL code

To build a simple assembler DLL, your assembler routine must conform to Language Environment conventions. To do this, begin by using the Language Environment macros CEEENTRY and CEETERM. The EXPORT= keyword parameter on the CEEENTRY 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 “Assembler macros” on page 399.

*Figure 22 on page 48 shows how to use Language Environment macros to create an Assembler DLL. The CEEENTRY 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 CEEENTRY 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, "DllVar" (initial value = 123) and "DllStr" (initial value is the C string "Hello World"). When the exported function "dllfunc" is called, it sets "DllVar" to 456 and truncates the "DllStr" C string to "Hello".*
Dynamic link libraries (DLLs)

Compiling your DLL code

For C source, compile with the DLL compiler option. When you specify the DLL compiler option, the compiler generates special code when calling functions and referencing external variables. Even if a simple application or DLL does not reference any imported functions or imported variables from other DLLs, you should specify the DLL compiler option. Compiling an application or DLL as DLL code eliminates the potential compatibility problems that may occur when binding DLL code with non-DLL code.

Compiling your C source with the XPLINK compiler option will automatic generate DLL-enabled code, so in this case the DLL compiler option is not necessary.

For C++ source, compile as you would any C++ program.

For COBOL source code that defines DLLs, compile with the RENT, DLL and EXPORTALL compiler options. For source code that only references DLLs, compile with the RENT, DLL, and NOEXPORTALL compiler options. An alternative to the DLL compiler option for Enterprise COBOL V6, you can use the CallInterface DLL.

For Enterprise PL/I source, you must compile with the RENT option.

For Assembler source, you must use the GOFF option.

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

Binding your DLL code

Use the DLL support in the DFSMS binder, rather than the linkage editor, for linking DLL applications. Note that binder-based DLLs must reside in PDSEs, rather than PDS data sets. If a DLL must reside in a PDS
Dynamic link libraries (DLLs)

load library, the application must be prelinked with the Language Environment prelinker before standard linkage editing. See Appendix A, “Prelinking an application,” on page 481 for further details. 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(PM3) 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
```

If the code in the DLL was compiled with the XPLINK compiler option, specify:

```
-W 1,DLL,XPLINK
```

- 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 and z/OS MVS Program Management: Advanced Facilities for further details.

Binding C

When binding the C object module as shown in Figure 19 on page 46, the binder generates the following definition side-deck:

```
IMPORT CODE 'BASICIO'   bopen
IMPORT CODE ,BASICIO,   bclose
IMPORT CODE ,BASICIO,   bread
IMPORT CODE ,BASICIO,   bwrite
IMPORT DATA ,BASICIO,   berror
```

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

**Note:**

1. You should also provide a header file that contains the prototypes for exported functions and external variable declarations for exported variables.
2. Sidedecks 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 about binding C, see z/OS XL C/C++ User's Guide.

Binding C++

When binding the C++ object modules shown in Figure 20 on page 46, the binder generates the following definition side-deck:

```
IMPORT CODE ,TRIANGLE, getarea__8triangleFv
IMPORT CODE ,TRIANGLE, getperim__8triangleFv
IMPORT CODE ,TRIANGLE, __ct__8triangleFv
```

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

**Note:**

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 COBOL**

When binding a module that contains COBOL programs compiled with the DLL and EXPORTALL compiler options, the binder generates a definition side-deck. If there are programs in the module that you do not want to make available with DLL linkage, you can edit the definition side-deck to remove programs that you do not want to export.

**Binding Enterprise PL/I**

The considerations for binding Enterprise PL/I are the same as for binding C++ in “Binding C++” on page 49.

**Binding Assembler**

When binding the Assembler object module as shown in Figure 22 on page 48, the binder generates the following definition side-deck:

```plaintext
IMPORT CODE,'ADLLBEV2','dllfunc'
IMPORT DATA,'ADLLBEV2','DllStr'
IMPORT DATA,'ADLLBEV2','DllVar'
```

The Assembler DLL support requires use of the binder.

**Building a simple DLL application**

A simple DLL application contains object modules that are made up of only DLL-code. The application may consist of multiple source modules. Some of the source modules may contain references to imported functions, imported variables, or both.

It is not necessary for DLL applications to be reentrant. However, for some compilers it is necessary to compile code that references DLLs with the RENT option in order to provide support for the DLL call mechanism.

To use a load-on-call DLL in your simple 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 imported variables must use the Language Environment macros.

- **Compiling your DLL application code**
  
  Compile your C source files normally. A C++ application is always DLL code.

  Compile your COBOL source files with the following compiler options:
  - DLL
  - RENT
- NOEXPORTALL
  - Compile your Enterprise PL/I source files with the RENT option.
  - Assembler DLL Application source files must be assembled using the G0FF option.
- Binding your DLL application code
  - The binder option CASE(MIXED) is required when binding DLLs applications that use mixed-case exported names.
  - The binder options RENT, DYNAM(DLL), and COMPAT(PM3) 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 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.WSA during program execution.

  The following code fragment illustrates how a C++ application can use the TRIANGLE DLL described in "Writing your C++ DLL code" on page 46. Compile normally and bind with the definition side-deck provided with the TRIANGLE DLL.

```c
extern int getarea(); /* function prototype */
main() {
  ...
  getarea();       /* imported function reference */
  ...
}
```
The following COBOL code sample illustrates how a simple COBOL-only DLL application (A1C4DL01) calls a COBOL DLL (A1C4DL02):

```cobol
CBL PGMNAME(LONGMIXED),DLL,RENT
IDENTIFICATION DIVISION.
PROGRAM-ID. 'A1C4DL01'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
01 TODAYS-DATE-YYYYMMDD PIC 9(8).
PROCEDURE DIVISION.
   Display 'A1C4DL01: Entered'
   MOVE FUNCTION CURRENT-DATE(1:8) TO TODAYS-DATE-YYYYMMDD
   Call 'A1C4DL02' using todays-date-yyyymmdd
   Display 'A1C4DL01: All done'
   GOBACK
.
CBL PGMNAME(LONGMIXED),DLL,EXPORTALL,RENT
IDENTIFICATION DIVISION.
PROGRAM-ID. 'A1C4DL02'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 TODAYS-DATE-YYYYMMDD PIC 9(8).
PROCEDURE DIVISION using todays-date-yyyymmdd.
   Display 'A1C4DL02: Todays date is ' todays-date-yyyymmdd
   GOBACK
.
```

Figure 23: COBOL DLL application calling a COBOL DLL
The following code fragment illustrates how an Assembler routine can use the ADLLBEV2 DLL described in “Writing your Language Environment-conforming assembler DLL code” on page 47. Assemble and bind with the definition side-deck provided with the ADLLBEV2 DLL.

```assembly
DLLAPPL CEEENTRY MAIN=YES,PPA=DLLPPA
*    Symbolic Register Definitions and Usage
R8   EQU   8             Work register
R9   EQU   9             Work register
R15  EQU   15            Entry point address
*    WTO   'ADLABIV4: Calling imported function dllfunc',ROUTCDE=11
*    CEEPCALL dllfunc,MF=(E,)
*    WTO   'ADLABIV4: Getting address of imported var DllVar', ROUTCDE=11
*    CEEPLDA DllVar,REG=9
*    * Set value of imported variable to 789
*    LA    R8,789
ST    R8,0(,R9)
*    WTO   'ADLABIV4: Done.',ROUTCDE=11
*    SR    R15,R15
RETURN DS   0H
CEETERM RC=(R15),MODIFIER=0
*    *    CEEPDDA DllVar,SCOPE=IMPORT
DLLPPA CEEPPA
LTORG
CEEDSA
CEECAA
END      DLLAPPL
```

Figure 24: Assembler DLL application calling an assembler DLL

See Figure 25 on page 54 for a summary of the processing steps required for the application and related DLLs.

Creating and using DLLs

Figure 25 on page 54 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 compilers 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.
Figure 25: Summary of DLL and DLL application preparation and usage

DLL restrictions

Consider the following restrictions when creating DLLs and DLL applications:

- The entry point for a DLL must be in a program compiled with a Language Environment-conforming compiler that includes DLL support capability. Otherwise, Language Environment issues an error and terminates the application.

- DLLs must be REENTRANT. Be sure to specify the RENT option when you bind your code. Unpredictable results will occur if you link-edit a DLL as NORENT. One possible symptom you may see that indicates the DLL was link-edited 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.

The AMODE of a DLL application must be the same as the AMODE of the DLL that it calls.

DLL facilities are not available:
- Under MTF, CSP or SP C
- To application programs with `main()` written in PL/I that dynamically call C functions. (This restriction does not apply to Enterprise PL/I)

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.

If a fetched C module is compiled as a DLL, it can import variables and functions from the other DLL modules, but it cannot export variables or functions.

A COBOL dynamic call cannot be made to a load module that is a DLL.

A COBOL dynamic call cannot be made to a COBOL for OS/390 & VM program that is compiled with the DLL compiler option.

COBOL data declared with the EXTERNAL attribute are independent of DLL support; these data items are managed by the COBOL runtime environment and are accessible by name from any COBOL program in the run-unit that declares them, regardless of whether the programs are in DLLs or not.

In particular, the facilities for exporting and importing external variables from DLLs implemented in C/C++ do not apply to COBOL external data.

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 `d11load()` 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 `dlclose()` 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 `dlclose()` (Language Environment will free them when the enclave is terminated).
- Have the `main()` function call `dllfree()` or `dlclose()` 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 topic 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 the LPA or ELPA. When the DLL resides in a PDSE, the dynamic LPA services should be used. Installing in the LPA/ELPA 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 z/OS UNIX file system 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 z/OS UNIX file system first. If your DLL is a data set member, you can avoid searching the 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 POSIX(OFF), Language Environment tries to load your DLL from a data set. If your DLL is a z/OS UNIX file, you can avoid searching a data set. To direct a DLL search to the z/OS UNIX file system, 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 z/OS UNIX file system. If you specify the wrong case for your DLL that resides in the z/OS UNIX file system, 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.

• The suboption NOCALLBACKANY of the C compiler option DLL is more efficient than the CALLBACKANY suboption. The CALLBACKANY option calls a Language Environment routine at runtime. This runtime service enables direct function calls. Direct function calls are function calls through function pointers that point to actual function entry points rather than function descriptors. The use of CALLBACKANY will result in extra overhead at every occurrence of a call through a function pointer. This is unnecessary if the calls are not direct function calls.

**Building complex DLLs**

Before you attempt to build complex DLLs it is important to understand the differences between the terms DLL, DLL code, and DLL application, as described in “DLL concepts and terms” on page 37.

Note that not all source files that make up a DLL application have to be compiled with the DLL option. However, source files that reference exported functions and exported global variables must be compiled with the DLL option.

A key characteristic of a complex DLL or DLL application is that linking DLL code with non-DLL code creates it. The following are reasons you might compile your code as non-DLL:

1. Source modules do not use C or C++.
2. To prevent problems which occur when a non-DLL function pointer call uses DLL code. This problem takes place when a function makes a call through a function pointer that points to a function entry rather than a function descriptor.

For more information on building complex DLLs, see z/OS XL C/C++ Programming Guide.
Chapter 5. Link-editing, loading, and running under batch

You process an 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
- Link-editing an application
- Running an application

**Note:** The terms in this topic having to do with linking (bind, binding, link, link-edit, and so forth) refer to the process of creating an 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. For linkage editor-specific information, see *z/OS MVS Program Management: User’s Guide and Reference* and *z/OS MVS Program Management: Advanced Facilities*.

IBM-supplied cataloged procedures allow you to compile, link-edit or load, and run an application without supplying all the job control language (JCL) required for a job step. For information about cataloged procedures, see Chapter 8, “Using IBM-supplied cataloged procedures,” on page 87. 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 link-editing, loading, and running Language Environment-conforming applications under batch. For detailed information about link-editing, see *z/OS MVS Program Management: User’s Guide and Reference* and *z/OS MVS Program Management: Advanced Facilities*. For information about the Language Environment prelinker, see Appendix A, “Prelinking an application,” on page 481.

Several Fortran and C library routines have identical names. If your application contains any Fortran or assembler routine that uses a Fortran library routine, see “Resolving library module name conflicts between Fortran and C” on page 15 to resolve any potential name conflicts.

TSO/E has its own section on link-editing, loading, and running (see Chapter 6, “Creating and executing programs under TSO/E,” on page 71).

*z/OS UNIX has its own section on link-editing, loading, and running C applications (see Chapter 7, “Creating and executing programs using z/OS UNIX System Services,” on page 79).

Basic link-editing and running under batch

This topic describes how to accept and to override the default Language Environment runtime options under MVS.

**Accepting the default runtime options**

To run an existing object module under batch and accept all of the default Language Environment runtime options, use the following sample JOB with the Language Environment-provided link-edit and run cataloged procedure CEEWLG (see “CEEWLG — Link and run a Language Environment conforming non-XPLINK program” on page 92 for more information). The CEEWLG procedure identifies the Language Environment libraries that your object module needs to link-edit and run; you do not need to explicitly identify these in your JCL.

There is also a cataloged procedure, CEEXLR, for XPLINK. See “CEEXLR — Link and run a Language Environment conforming XPLINK program” on page 94.
## Overriding the default runtime options

In Figure 27 on page 58, an object module called MYPROG is created and run using the cataloged procedure CEEWLG. The code in the example overrides the Language Environment defaults for the RPTOPTS and MSGFILE runtime options.

```clike
//CEEWLG JOB
//--*
//LINKGO EXEC CEEWLG,
//-- PARM.GO='RPTOPTS(ON),MSGFILE(OPTRPRT)'/
//--*
//LKED.SYSIN DD DSN='userid.MYLIB.OBJLIB(MYPROG)',...DISP=SHR
//--*
//GO.OPTRPRT DD SYSOUT=A
//--*

Figure 27: Overriding the default runtime options under batch
```

The trailing slash after the runtime options is required for C, Fortran, PL/I and for COBOL users who have specified the CBLOPTS(OFF) runtime option. For COBOL users who have specified the CBLOPTS(ON) runtime option at installation (see z/OS Language Environment Programming Reference), the slash should go before the runtime options, as in Figure 28 on page 58.

```clike
: : // PARM.GO='/RPTOPTS(ON),MSGFILE(OPTRPRT)'

Figure 28: Overriding the default runtime options for COBOL
```

## Specifying runtime options with the CEEOPTS DD card

Language Environment supports the ability to provide additional runtime 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.

In the Language Environment Runtime Options report, when an option was last set in the CEEOPTS DD card, DD:CEEOPTS will be used in the "LAST WHERE SET" column.

The CEEOPTS DD is ignored under CICS, SPC, and for programs invoked using one of the exec family of functions.

The general form for specifying runtime options with the CEEOPTS DD card is:

```clike
//CEEWLG JOB
//--*
//LINKGO EXEC CEEWLG
//-- LKED.SYSIN DD DSN='userid.MYLIB.OBJLIB(MYPROG)',...DISP=SHR
//-- GO.OPTRPRT DD SYSOUT=A
//-- GO.CEEOPTS DD *

Figure 26: Accepting the default runtime options under batch
```
Specifying runtime options in the EXEC statement

If the first program in your application is Language Environment-conforming or was compiled by a pre-Language Environment compiler supported by Language Environment, 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 program PROGRAM1, specify the following:

```
//GO1   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 104. 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.

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.

For Enterprise PL/I for z/OS and PL/I for MVS & VM, runtime options can be passed in your JCL if a PROCEDURE statement includes the OPTIONS(MAIN) clause. If the PROCEDURE statement specifies OPTIONS(MAIN NOEXECOPS), then runtime options cannot be passed in your JCL. Note that no PL/I compiler has an NOEXECOPS or EXECOPS compiler option, but they have the equivalent function by the specification of NOEXECOPS along with OPTIONS(MAIN).

Providing link-edit input

Input to the link-edit process can be:

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

Figure 29 on page 60 shows the basic batch link-edit process for your application.
Writing JCL for the link-edit 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 link-edit process.

For a description of the IBM-supplied cataloged procedures that include a link-edit step, see Table 15 on page 88.

• Invoking with the EXEC statement.

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

```plaintext
//LKED EXEC PGM=HEWL
```

• 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 executable program produced by the link-edit process, specify:

```plaintext
//LKED EXEC PGM=HEWL,PARM='MAP'
```

• Required DD statements.

The link-edit process requires three standard data sets. You must define these data sets in DD statements with the ddnames SYSLIN, SYSLMOD, and SYSPRINT. If the linkage editor is being used then an additional data set must be defined with ddname SYSUT1. The required data sets and their characteristics are shown in Table 8 on page 61.
### Table 8: Required data sets used for link-editing

<table>
<thead>
<tr>
<th>ddname</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSLIN</td>
<td>Input</td>
<td>Primary input to the link-edit 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 link-edit 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 64).</td>
</tr>
<tr>
<td>SYSLMOD</td>
<td>Output</td>
<td>The data set where output (executable program) from the link-edit process is stored.</td>
</tr>
<tr>
<td>SYSPRINT</td>
<td>Output</td>
<td>SYSPRINT defines the location for the listing that includes reference tables for the executable program. Output from the link-edit process:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Diagnostic messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Informational messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Module map</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Cross-reference list</td>
</tr>
<tr>
<td>SYSUT1</td>
<td>Utility</td>
<td>A data set used by the linkage editor as a temporary workspace (the data set must be on a direct access device). This data set is not required for the binder.</td>
</tr>
</tbody>
</table>

- 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 executable programs as additional input to the link-edit process. These data set names and their characteristics are shown in Table 9 on page 62.

---

**Running under batch**

Link-editing, loading, and running under batch 61
Table 9: Optional data sets used for link-editing.

<table>
<thead>
<tr>
<th>ddname</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
</table>
| SYSLIB1   | Library  | Secondary input to the linkage editor 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 linkage editor to resolve external symbols left undefined after all the primary input has been processed. The automatic call library can include:  
  - Libraries that contain object modules, with or without linkage editor control statements  
  - Libraries that contain executable programs  
  - The libraries that contain the Language Environment resident routines, such as SCEELKED, SCEELKEX, SCEEOBJ, and SCEECPP (for a description of these data sets see “Planning to link-edit and run” on page 7). SYSLIB is input to the linkage editor only if the CALL=NO link-edit option is not in effect (see Table 10 on page 65, in z/OS MVS Program Management: User’s Guide and Reference and z/OS MVS Program Management: Advanced Facilities, or in z/OS TSO/E Command Reference for more information). You can also identify secondary input to the linkage editor with the INCLUDE statement. 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 SCEELKED library; the data set name could be CEE.SCEELKED. If you are unsure where SCEELKED has been installed at your location, contact your system administrator. This data set must be specified in the SYSLIB statement in your JCL. 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:  

//SYSLIB DD DNAME=CEE.SCEELKED,DISP=SHR

User-specified2 Input You can use ddnames to get additional executable programs and object modules.

Notes:
1  
  Required for library runtime routines  
2  
  Optional data set  
• Examples of link-edit JCL.

A typical sequence of job control statements for link-editing an object module (compiled NOXPLINK) into an executable program is shown in Figure 30 on page 63. The NAME linkage editor control statement in the figure puts PROGRAM1 in USER.LOADLIB with the member name PROGRAM1.
Figure 30: Creating a non-XPLINK executable program under batch

A similar sequence of job control statements for link-editing an XPLINK object module is shown in Figure 31 on page 63.

Figure 31: Creating an XPLINK 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 (for example, SCEELKED/SCEELKEX for non-XPLINK applications, and SCEEBND2 for XPLINK 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 linkage editor, 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 linkage editor, 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.

- Link-editing multiple object modules.

When an Enterprise PL/I for z/OS or PL/I for MVS & VM main procedure is link-edited with other object modules produced by the Enterprise PL/I for z/OS or the PL/I for MVS & VM compilers, the entry point of the resulting executable program is resolved to the external symbol CEESTART. This happens automatically because the CEESTART CSECT is generated first in the object module and is specified in the END statement of the object module. Runtime errors occur if the executable program entry point is forced to some other symbol by use of the linkage editor ENTRY control statement.

If an Enterprise PL/I for z/OS or PL/I for MVS & VM main procedure is link-edited with object modules produced by other language compilers or by assembler, and is the first module to receive control, the user must ensure that the entry point of the resulting executable program is resolved to the external symbol CEESTART. This happens automatically if the Enterprise PL/I for z/OS or PL/I for MVS & VM main
procedure is first in the input to the linkage editor. Runtime errors occur if the executable program entry point is forced to some other symbol by use of the linkage editor ENTRY control statement.

Alternatively, the following linkage editor ENTRY control statement can be included in the input to the linkage editor:

```
ENTRY CEESTART
```

**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 executable programs that you want included in the output executable program. Figure 32 on page 64 contains an example of how to link-edit the CEEUOPT CSECT with your application. In the example, CEEUOPT is used to establish application runtime option defaults; see Chapter 9, “Using runtime options,” on page 101 for more information.

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

Figure 32: Using the INCLUDE linkage editor 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 33 on page 64 the LIBRARY statement is used to resolve the external reference PROGRAM2 from the library described in the TESTLIB DD statement.

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

Figure 33: Using the LIBRARY linkage editor control statement
```

Data sets specified by the INCLUDE statement are incorporated as the linkage editor 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.

**Link-edit options**

SYSLMOD and SYSPRINT are the data sets used for output. The output varies, depending on the options you select, as shown in Table 10 on page 65.

64  z/OS: Language Environment Programming Guide
Table 10: Selected link-edit options

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>**XREF</td>
<td>NOXREF**</td>
</tr>
<tr>
<td>**LIST</td>
<td>NOLIST**</td>
</tr>
<tr>
<td>**NCAL</td>
<td>CALL**</td>
</tr>
<tr>
<td>**PRINT</td>
<td>NOPRINT**</td>
</tr>
<tr>
<td>**MAP</td>
<td>NOMAP**</td>
</tr>
<tr>
<td>**RENT</td>
<td>NORENT**</td>
</tr>
</tbody>
</table>

You always receive diagnostic and informational messages as the result of link-editing, 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 link-editing. See “Writing JCL for the link-edit process” on page 60 for more information.

For more information about link-edit options, see z/OS MVS Program Management: User’s Guide and Reference and z/OS MVS Program Management: Advanced Facilities.

Loading your application using the loader

Your input to the loader can be:

- One or more HLL object modules.
- One or more previously link-edited HLL load modules that you want to combine into a single load module.
- A combination of both.

If you include any linkage control statements (such as LIBRARY or INCLUDE) as input to the loader, an informational error message is printed in the output listing only if you have a SYSLOUT DD statement in your input JCL (see Figure 35 on page 68). Otherwise, the linkage control statements are ignored.

In basic loader processing, as shown in Figure 34 on page 66, the loader accepts data from its primary input source, a data set defined by the SYSLIN DD statement. This data set is the object module produced by the compiler. The loader uses the external symbol dictionary in SYSLIN to determine whether the object module includes any external references that have no corresponding external symbols in SYSLIN.
The loader searches the automatic call library, SYSLIB, (as shown in Figure 34 on page 66) for the routines in which the external symbols are defined and includes them in the load module if they exist. If all external references are resolved, the load module is executed.

Your application cannot be executed until the appropriate runtime routines have been included.

### Writing JCL for the loader

If you use cataloged procedures (rather than supply all of the JCL required for a job step that invokes the loader), you should be familiar with JCL statements for the loader so you can make the best use of the loader and, if necessary, override the statements of the cataloged procedure.

The following sections describe the basic JCL statements for loading. For a description of the IBM-supplied cataloged procedures that include a loader step, see Table 15 on page 88.

#### Invoking the loader with the EXEC statement

Use the EXEC statement to invoke the loader. The EXEC statement to invoke the loader is:

```
//GO EXEC PGM=LOADER
```

#### Using the PARM parameter for loader options

Use the PARM parameter of the EXEC statement to specify loader options in your JCL for loader processing. For example, if you want your application to run even if abnormal conditions are detected, and you want a mapping of the executable program, specify the following:

```
//GO EXEC PGM=LOADER, PARM='MAP,LET'
```
Requesting loader options

Table 11 on page 67 shows you which options you can specify as PARM parameters when running the loader.

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP</td>
<td>NOMAP</td>
</tr>
<tr>
<td>LET</td>
<td>NOLET</td>
</tr>
<tr>
<td>CALL</td>
<td>NOCALL</td>
</tr>
<tr>
<td>EP = name</td>
<td>Specifies the name of the entry point of the application being loaded.</td>
</tr>
<tr>
<td>PRINT</td>
<td>NOPRINT</td>
</tr>
<tr>
<td>RES</td>
<td>NORES</td>
</tr>
<tr>
<td>SIZE = size</td>
<td>Specifies the amount of storage allocated by loader processing; size includes the size of your executable program.</td>
</tr>
</tbody>
</table>

When you run the loader, you can request the options shown in Table 11 on page 67 using the PARM parameter of the EXEC statement. For more information about specifying and using loader options, see z/OS MVS Program Management: User’s Guide and Reference and z/OS MVS Program Management: Advanced Facilities.

Passing parameters through the loader

Code the PARM parameter as follows:

```
PARM='[loader-options] [/runtime-options] [/pgmparm]'
```

where loader-options is a list of loader options, runtime-options is a list of runtime options, and pgmparm is a parameter string passed to the main routine of the application to run. The following examples refer to the routine parameter as PP. If you specify NOEXECOPS on the main routine, you must omit the slash in front of pgmparm.

If you specify loader options and either runtime options or a routine parameter (or both) in the PARM parameter, the loader options are given first and are separated from the runtime options or routine parameter by a slash. If there are loader options but no runtime options or routine parameters, the slash is omitted. If there are only runtime options or routine parameters, you must code the slash or slashes. If there is more than one option, separate the option keywords by commas.

The PARM field can have one of the following formats:

- If you use the special characters / or =, you must enclose the field in single quotes. For example:

  ```
  PARM='MAP, EP=FIRST/RPTOPTS(ON)/PP'
  PARM='MAP, EP=FIRST'
  PARM='//PP'
  ```
If you do not use the / or = characters, and there is more than one loader option, you must enclose the options in parentheses. For example:

```
PARM=(MAP,LET)
```

If you do not use the / or = characters, and there is only one loader option, neither quotes nor parentheses are required. For example:

```
PARM=MAP
```

Using DD statements for the standard loader data sets

The loader always requires one standard data set, defined by the SYSLIN DD statement. Three other standard data sets are optional, and, if you use them, you must define them in DD statements with the names SYSLOUT, SYSPRINT, and SYSLIB. The four data set names and characteristics are shown in Table 12 on page 68.

<table>
<thead>
<tr>
<th>ddname</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSLIN</td>
<td>Input</td>
<td>Primary input data (normally the compiler output)</td>
</tr>
<tr>
<td>SYSLOUT</td>
<td>Output</td>
<td>Loader messages and module map listing</td>
</tr>
<tr>
<td>SYSPRINT</td>
<td>Output</td>
<td>Runtime messages and problem output listing</td>
</tr>
<tr>
<td>SYSLIB</td>
<td>Library</td>
<td>Automatic call library</td>
</tr>
</tbody>
</table>

Figure 35 on page 68 is an example of the general job control procedure for creating and running an executable program under batch.

```
//STEP1    EXEC PGM=LOADER,PARM='MAP,LET'
//STEPLIB  DD DSN=CEE.SCEERUN,DISP=SHR
//         DD DSN=CEE.SCEERUN2,DISP=SHR
//SYSLIB   DD DSN=CEE.SCEELKED,DISP=SHR
//         DD DSN=USER.LOADLIB,DISP=SHR
//SYSLIN   DD DSN=USER.OBJLIB(PROGRAM1),DISP=SHR
//SYSLOUT  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
```

Figure 35: JCL for creating an executable program

Running an application under batch

Under batch, you can request the execution of an 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, either the Link Pack Area (LPA) or the Extended Link Pack Area (ELPA).

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.LOADLIB for the routine PROGRAM1 and USER.LOADLIB2 for the routine PROGRAMA:

```
//JOB8  JOB  DAVE,MSGLEVEL=(2,0)
//STEP1  EXEC PGM=PROGRAM1
//STEPLIB DD   DSNAME=USER.LOADLIB,DISP=SHR
//*
//STEP2  EXEC PGM=PROGRAMA
//STEPLIB DD   DSNAME=USER.LOADLIB2,DISP=SHR
```

With the ddname JOBLIB immediately after the JOB statement in your JCL. This library is searched before the system libraries. If any 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.LOADLIB for the member PROGRAM1, reads the member into storage, and executes it.

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

Specifying runtime options under batch

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 runtime options in any of the following places (for additional information about the ways to specify runtime options, see “Methods available for specifying runtime options” on page 101):

- In the CEEROPT CSECT, where region-level defaults are specified. For more information, see z/OS Language Environment Customization.
• In the CEEUOPT CSECT where user-supplied default options are located. For more information, see “CEEXOPT invocation for CEEUOPT” on page 106).

• In the CEEPROMxx parmlib member where system-level defaults are specified. For more information, see z/OS Language Environment Customization.

• #pragma runopts in C/C++ source code (for more information, see “Methods available for specifying runtime options” on page 101).

• In a PLIXOPT string in PL/I source code (for more information, see “Methods available for specifying runtime options” on page 101).

• 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 assembler user exit (for more information, see “CEEBXITA assembler user exit interface” on page 378).

• In the _CEE_RUNOPTS environment variable, when your application is running under z/OS UNIX and is invoked by one of the exec family of functions.
Chapter 6. Creating and executing programs under TSO/E

Under TSO/E, you process an application by compiling and link-editing the programs that make up the application, and then running the application.

The compiler produces an object module; the link-edit process takes the object module and produces an executable program. You can link-edit and run your application as separate steps (LINK and CALL) or you can link-edit and run your application as a single step (LOADGO).

Note: Several Fortran and C library routines have identical names. If your application contains any Fortran or assembler routine that uses a Fortran library routine, see “Resolving library module name conflicts between Fortran and C” on page 15 to resolve any potential name conflicts.

Basic link-editing and running under TSO/E

This topic describes how to accept and to override the default Language Environment runtime options under TSO/E.

Accepting the default runtime options

Use the LOADGO command to run an existing NOXPLINK-compiled object module under TSO/E and to accept the default Language Environment runtime options. See “Specifying runtime options and program arguments” on page 104 for more information on using runtime options. For example, the command

\[
\text{LOADGO ('userid.MYLIB.OBJLIB(MYPROG)') LIB ('CEE.SCEELKED')}
\]

does the following:

- Takes the existing object module MYPROG from the object library in which you have it stored
- Links in the Language Environment (text) link library SCEELKED
- Runs the new executable program

Overriding the default runtime options

The following example overrides the Language Environment defaults for the RPTOPTS and MSGFILE runtime options, and loads and runs the XPLINK-compiled program MYPROG:

\[
\text{LOADGO ('userid.MYLIB.OBJLIB(MYPROG)') 'RPTOPTS(ON), MSGFILE(OPTRPRT)'/ LIB ('CEE.SCEEBIND')}
\]

The Language Environment data sets SCEELKED link library, SCEEBND2 link library, and the SCEERUN dynamic library (needed before you can run your executable program) could have been installed with a different high-level qualifier than CEE. Check with your system administrator for the correct names.

The LOADGO command is described in detail in “Loading and running using the LOADGO command” on page 75.

Specifying runtime options with the CEEOPTS DD card

Language Environment supports the ability to provide additional runtime 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.

In the Language Environment Runtime Options report, when an option was last set in the CEEOPTS DD card, DD:CEEOPTS will be used in the “LAST WHERE SET” column.
Link-editing and running

The LINK command link-edits a compiled external procedure or combines two or more procedures to form a single executable program. You can run an executable program by using the CALL command. Use the LINK-and-CALL method of processing when you want to:

- Keep a copy of the executable program in an external data set
- Link-edit two or more external procedures to form a single executable program
- Run a program repeatedly, without changing the source code

To run a compiled program without keeping a copy of the executable program, use the LOADGO command.

Link-editing your application using the LINK command

The LINK command invokes the linkage editor, which converts one or more object modules into an executable program suitable for execution. Later, you can run the executable program using the CALL command (see “Using the CALL command to run your application” on page 74). The general form of the LINK command is:

```
LINK (data-set-list)
```

Specifies the names of the data sets containing the object modules to be link-edited. The variable `data-set-list` must contain at least one object module, but can also contain binder control statements. If you have only one name, you can omit the parentheses. If there are several names, you must separate them by commas or blanks within the parentheses. The rules for positioning control statements in relation to object modules are the same as for batch mode. If you specify a simple data set name, the system assumes the descriptive qualifier `OBJ`; that is, the data set name is of the form `userid.data-set-name.OBJ`.

```
LOAD (data-set-name)
```

Specifies the name of the data set to contain the executable program generated by the link-edit process. If you specify a simple name, the system adds the user-identification qualifier and the descriptive qualifier `LOAD (userid.data-set-name.OBJ)`, and uses that as the data set name. The resulting executable program must be stored as a member in a PDS or PDSE. If you do not supply a member name, the executable program is placed in member `TEMPNAME` of the `userid.data-set-name.OLOAD` data set. If you do not specify LOAD, `userid.LOAD` is used.

```
LIB (data-set-list)
```

Specifies the names of data sets that contain user-supplied modules that you want to be link-edited by the automatic library call facility.

The appropriate link-edit libraries, including the Language Environment link-edit libraries, must be specified. See “Planning to link-edit and run” on page 7 for a description of the Language Environment link-edit libraries.

```
options
```

Specifies a list of link-edit processing options. You must separate the options with a valid delimiter such as a comma or blank. Table 14 on page 77 contains a partial listing of available link-edit options.

The following example shows how to:
• Link-edit two object modules named PROGRAM1 and CEEUOPT. CEEUOPT can be used to establish programmer runtime option defaults. See Chapter 9, “Using runtime options,” on page 101 for more information.

• Load the resulting executable program in member PROGRAM1 in the library USER.LOADLIB.

• Specify the Language Environment library CEE.SCEELKED as the automatic call library for a non-XPLINK application.

• Generate a mapping of the executable program that is run by using the MAP option.

• Direct the linkage editor listing to the terminal by using the PRINT(*) option.

```
LINK ('USER.OBJLIB(PROGRAM1)','USER.OBJLIB(CEEUOPT)')
LOAD('USER.LOADLIB(PROGRAM1)')
LIB ('CEE.SCEELKED') MAP PRINT(*)
```

For more information about using the TSO/E LINK command and its options, see z/OS TSO/E Command Reference.

**Using CMOD CLIST to invoke the TSO/E LINK command**

You can use CMOD to build C modules or C ILC applications where C is the main routine. CMOD invokes the TSO/E LINK command by passing all CMOD parameters to that command. Any parameters not passed from CMOD have the normal LINK command default values. The CMOD CLIST resides in CEE.SCEECLST. The CMOD CLIST cannot be used to link XPLINK applications.

**OBJ**
- Specifies input object data set names.

**LOPT**
- Specifies a string of linkage editor options.
LIB
Specifies libraries that you want to use to resolve external references. These libraries are appended to
the default C library functions.

LOAD
An output data set name. If you do not specify an output data set name, a name is generated for you.
The name generated by the CLIST consists of your user prefix followed by LOAD(TEMPNAME).

Table 13 on page 74 shows CMOD calls and their corresponding results:

<table>
<thead>
<tr>
<th>Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmod obj(myobj)</code></td>
<td><code>link userid.myobj lib(cee.sceelked)</code></td>
</tr>
<tr>
<td><code>cmod obj(myobj)</code></td>
<td><code>link userid.myobj lib(userid.mylib cee.sceelked) rmode(24)</code></td>
</tr>
<tr>
<td><code>load(myload)</code></td>
<td><code>load(myload) amode(24)</code></td>
</tr>
</tbody>
</table>

Possible error messages are:
- CMOD: NO INPUT OBJECT DECK SPECIFIED (rc=16)
- CMOD: ERROR WITH INPUT OBJECT DECK (rc=16)
- Any error messages generated by LINK

Using the CALL command to run your application

The TSO/E CALL command loads and executes a specified executable program. To run an application
successfully, the SCEERUN and SCEERUN2 dynamic libraries must be either in the link-list concatenation,
or in a STEPLIB in the TSO/E logon procedure. As an alternative, the TSO/E Dynamic STEPLIB Facility
(Program Offering 5798-DZW) can be used to dynamically allocate SCEERUN and SCEERUN2 to the
execution environment. For more information about the TSO/E logon procedure, see z/OS Program
Directory in the z/OS Internet library (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/
zosInternetLibrary).

When you use the CALL command, you must also specify the ALLOCATE command to dynamically allocate
the data sets required by the application you intend to run. For more information, see z/OS TSO/E
Command Reference.

The general form of the CALL command is:
**data-set-name**
Specifies the data set that holds the executable program. If you specify the simple name of the data set, the system assumes the descriptive qualifier LOAD. If you do not specify a member name, the system assumes the name TEMPNAME.

You can also specify the member name of the data set that holds the executable program you plan to run, as indicated in the syntax diagram.

**program_parms**
A list of runtime options and program parameters passed to the main routine. Use a slash (/) to separate the runtime options and program parameters.

**ASIS**
Specifies that the program parameters are to be left in their original case. For C or C++, however, you must specify at least one lowercase character in the program parameter string for the case to be preserved, otherwise C or C or C++ lowercases the string.

For example, if you want to load and run member PROGRAM1 located in the data set USER.LOADLIB, and pass runtime options that generate storage and runtime options reports, specify the following:

```
CALL 'USER.LOADLIB(PROGRAM1) ' 'RPTSTG(ON),RPTOPTS(ON)' /
```

For a summary of formatting considerations for specifying runtime options, see “Specifying runtime options and program arguments” on page 104.

The EXECOPS option for C and C or 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. 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 both z/OS XL C/C++ and z/OS XL C++. If EXECOPS is specified, any runtime options specified in the CALL command are treated as program parameters.

**Note:** When using CALL to execute a program in the background under TSO/E with PGM=IKJEFT01, the ABTERMENC(ABEND) runtime option will not be processed correctly. To provide ABEND support under TSO/E, use PGM=IKJEFT1A or PGM=IKJEFT1B. For more information regarding these entry points, see z/OS TSO/E Customization.

**TSO/E parameter list format**
The TSO/E parameter list format differs depending on the language of the routine. Refer to Appendix D, “Operating system and subsystem parameter list formats,” on page 503 for language-specific considerations.

---

**Loading and running using the LOADGO command**
Use the LOADGO command to create an executable program in main storage and then run it. When the application has run, TSO automatically deletes the executable program created by LOADGO. The general form of the LOADGO command is:

```
LOADGO (data-set-list) 'program-parameter-string'
LIB (data-set-list) options
```

Creating and executing programs under TSO/E  75
LOADGO (data-set-list)
  Specifies the names of one or more object modules or executable programs that you want to load and run. If you have only one name, you can omit the parentheses. If you have several names, you must separate them by commas or blanks.

  The names can be data set names, names of members of data sets, or both.

program-parameter-string
  Specifies runtime options and program parameters to be passed to the executable program at run time. Use a slash (/) to separate the runtime options and parameters that are passed to the main routine in the executable program. The possible combinations are described in “Specifying runtime options and program arguments” on page 104.

LIB (data-set-list)
  Specifies the names of data sets containing user-supplied modules that you want the automatic library call facility to link-edit.

  You must also list the Language Environment resident library (SCEELKED for non-XPLINK applications or SCEEBIND for XPLINK applications). SCEELKED is installed into a data set with a high-level qualifier; for example, the name might be CEE.SCEELKED. If you are unsure of the name of the data set where SCEELKED has been installed at your location, contact your system administrator.

options
  Specifies a list of loader options. You must separate the options with a valid delimiter such as a comma or blank space. For a description of loader options, see Table 14 on page 77 and z/OS TSO/E Command Reference.

Allocating data sets under TSO/E

  When you use the LOADGO command under TSO/E, you must also specify the ALLOCATE command to dynamically allocate the data sets required by the application you intend to run. For more information, see z/OS TSO/E Command Reference.

Example of using LOADGO

  The following example shows how to:
  • Create an executable program using the object modules PROGRAM1 and CEEUOPT
  • Specify the runtime options to produce the runtime options report (RPTOPTS) and the storage report (RPTSTG)
  • Specify the Language Environment library CEE.SCEELKED as the automatic call library for a non-XPLINK application
  • Generate a mapping of the executable program
  • Direct the loader listing to the terminal

  LOADGO ('USER.OBJLIB(PROGRAM1)','USER.OBJLIB(CEEUOPT)')
  RPTOPTS(ON),RPTSTG(ON)/
  LIB ('CEE.SCEELKED') MAP PRINT(*)

To run an application successfully under TSO/E, the SCEERUN dynamic library must be either in the link-list concatenation, or in a STEPLIB in the TSO/E logon procedure. As an alternative, the MVS/TSO Dynamic STEPLIB Facility (Program Offering 5798-DZW) can be used to dynamically allocate SCEERUN to the TSO/E execution environment. For more information about the TSO/E logon procedure, see z/OS Program Directory in the z/OS Internet library (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary).

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. For C++, a user can specify either option in a #pragma runopts statement. For both C++
and z/OS XL C++ users, the options can be specified as compiler options. EXECOPS is the default for both z/OS XL C/C++ and z/OS XL C++. When EXECOPS is in effect, you can specify runtime options in the LOADGO command.

Link-edit and loader options

Table 14 on page 77 contains a partial listing of available link-edit and loader options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>NOCALL</td>
</tr>
<tr>
<td>SIZE(integer)</td>
<td>Specifies the size, in bytes, of the dynamic storage that the loader can use.</td>
</tr>
<tr>
<td>PRINT(data_set_name)</td>
<td>PRINT specifies the name of the data set that is used to contain the listing. You can direct output to the terminal by specifying PRINT(*).</td>
</tr>
<tr>
<td>MAP</td>
<td>NOMAP</td>
</tr>
<tr>
<td>LET</td>
<td>NOLET</td>
</tr>
<tr>
<td>RES</td>
<td>NORES</td>
</tr>
</tbody>
</table>

Using the iconv utility and ICONV CLIST for C/C++

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

The iconv utility can also perform bidirectional layout transformation (for example, shaping and reordering) on the data to be converted according to the two environment variables, _BIDION and _BIDIATTR. See the section on bidirectional language support in z/OS XL C/C++ Programming Guide for...
more information on bidirectional layout transformation. See the section on using environment variables in z/OS XL C/C++ Programming Guide for a description of _BIDION and _BIDIATTR.

See z/OS XL C/C++ User's Guide for information on the iconv utility.

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

The ICONV CLIST invokes the iconv utility to copy the input data set to the output data set and convert the characters from the input code page to the output code page.

Using the genxlt utility and GENXLT CLIST for C/C++

The genxlt utility reads character conversion information from an input file and writes the compiled version to an output file. The input file contains directives that are acted upon by the genxlt utility to produce the compiled version of the conversion table. See z/OS XL C/C++ User's Guide for more information on the genxlt utility.

The GENXLT CLIST invokes the genxlt utility to read the character conversion information and produce the conversion table. It then invokes the system linkage editor to build the executable program.

Running your application under TSO/E

You can run your TSO/E application in the following ways:

• Use LOADGO to create a module in main storage and then run it. For a description of the LOADGO command, see “Loading and running using the LOADGO command” on page 75.

• Use the CALL command to run an executable program that you have created using LINK. For a description of the LINK command, see “Link-editing your application using the LINK command” on page 72. For a description of the CALL command, see “Using the CALL command to run your application” on page 74.

• Run your application as a command processor.
Chapter 7. Creating and executing programs using z/OS UNIX System Services

The interface to the linkage editor for z/OS UNIX System Services (z/OS UNIX) C applications is the z/OS UNIX c89 utility or the cc utility, and for C++ applications it is the c++ utility. You can use them to compile and link-edit a z/OS UNIX C/C++ program in one step, or link-edit 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 these utilities, see z/OS UNIX System Services Command Reference.

Fortran applications are not supported under z/OS UNIX. When POSIX threading services are used, Fortran routines can only run in the initial process thread (IPT).

COBOL programs are supported under z/OS UNIX. See “Running COBOL programs under z/OS UNIX” on page 83 for more information.

Enterprise PL/I for z/OS has support for z/OS UNIX that is essentially the same as that of C++. Therefore everything in this topic that applies to C++ is also applicable to Enterprise PL/I for z/OS.

PL/I for MVS & VM routines are supported under z/OS UNIX. PL/I for MVS & VM routines can run in the IPT without any unique restrictions other than those described in the appropriate migration guide in the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735). PL/I for MVS & VM routines can run in the non-IPTs created by C/C++ routines with some restrictions. Limited PL/I – C/C++ ILC is supported in non-IPTs. See “Basic link-editing and running PL/I routines under z/OS UNIX with POSIX(ON)” on page 84 for more information.

PL/I MTF applications require z/OS UNIX services. PL/I MTF applications do not support ILC with C/C++ and must not invoke any z/OS UNIX services through an assembler program; otherwise the results are unpredictable.

Basic link-editing and running C/C++ applications under

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

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

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

See z/OS UNIX System Services Command Reference for information about 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 link-edit and create executable files

To link-edit a z/OS UNIX C/C++ application's object modules to produce an executable file, specify the c89 utility and pass it object modules (file.o z/OS UNIX 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.

To compile source files without link-editing 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 link-edit an XPLINK-compiled application object module to create the mymodx.out executable file in the current directory, specify:

  c89 -o mymodx.out -Wl,xplink usersource.o

- To link-edit an application object module to create the default executable file a.out in the working directory, specify:

  c89 usersource.o

- To link-edit an 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 usersource.o

- To link-edit several 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 usersrc.o ottrsrc.o //PGM.OBJ(PW...APP)

- To link-edit an 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) usersource.o

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

  c89 -o approg/lib/zinfo usersrc.c existobj.o
  //PGM.OBJ(PWAPP)

Running z/OS UNIX C/C++ application programs

This topic discusses the different ways you can run your z/OS UNIX C/C++ applications under z/OS.

z/OS UNIX application program environments

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

- z/OS UNIX shell
- TSO/E

You cannot directly call a z/OS UNIX application that resides in a z/OS UNIX file system 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 a z/OS UNIX file system because you cannot put a z/OS UNIX file name on the JCL EXEC statement. However, by using the BPXBATCH program, you can run a z/OS UNIX application that resides in an z/OS UNIX 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 a z/OS UNIX file system 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 C/C++ application executable file as a member in an MVS data set and want to place it in the z/OS UNIX file system, you can use the OPUTX or OGETX z/OS UNIX TSO/E commands to copy the member into a z/OS UNIX 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 z/OS UNIX files, see z/OS UNIX System Services User’s Guide.

### Restriction on using 24-Bit AMODE programs

You cannot run an AMODE(24) C/C++ application that resides in a z/OS UNIX file. Any programs you intend to run from the file system must be AMODE(31), problem program state, PSW key 8 programs. If you plan to run an AMODE(24) C/C++ program from within a z/OS UNIX application, make sure the executable program resides in a MVS PDS or PDSE member. Any new z/OS UNIX C/C++ applications you develop should be AMODE(31). XPLINK-compiled applications must be AMODE(31), and they and will force the ALL31 runtime option to ON.

### Running an MVS executable program from a z/OS UNIX shell

If your z/OS UNIX C/C++ 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 a z/OS UNIX file by using the 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 z/OS UNIX 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,
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. For example:

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

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

### Issuing a setup shell script from a shell

To run a z/OS UNIX shell script that sets up a 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](https://www.ibm.com/support/knowledgecenter/SSSTX8_2.4.0/com.ibm.zos.r24.doc/zos/unix/cmdref/chmod.html) for the format and description of `chmod`. After mode authority has been given, enter the script file name from the shell command line.

### Running an MVS batch z/OS UNIX C/C++ application file that is HFS-resident

To run a z/OS UNIX C/C++ executable application file from a z/OS UNIX file under MVS batch, invoke the IBM-supplied BPXBATCH program either from TSO/E, or by using JCL or a REXX EXEC (not batch). BPXBATCH performs an initial user login to run a specified program from the shell environment.

Before you invoke BPXBATCH, you must have the appropriate privilege to read from and write to z/OS UNIX files. You should also allocate STDOUT and STDERR 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 BPXBATCH program syntax and its use, and an example of running shell utilities under MVS batch using the BPXBATCH program, see [z/OS UNIX System Services Command Reference](https://www.ibm.com/support/knowledgecenter/SSSTX8_2.4.0/com.ibm.zos.r24.doc/zos/unix/cmdref/bpxbatch.html).

### Running POSIX-enabled programs outside the z/OS UNIX shells

#### Invoking BPXBATCH from TSO/E

You can invoke BPXBATCH 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:

```rexx
/* 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 BPXBATCH. When the REXX EXEC completes, the STDOUT and STDERR allocated files are deleted.

**Invoking BPXBATCH using JCL**

To invoke BPXBATCH using JCL, submit a job that executes an application program and allocates the standard files using DD statements. For example, 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:

```plaintext
//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 z/OS UNIX file system. 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.

```plaintext
/* REXX */
RC = SYSCALLS('ON')
If RC<0 | RC>4 Then Exit RC
Address SYSCALL
fstdout = 'fstdout'
stdin = 'stdin'
'debug object file stdout 0 _32770
'open' fstdout O_RDWR+O_TRUNC+O_CREAT 700
stdout = RETVAL
'debug object file stderr 0 _32770
'open' stderr O_RDWR+O_TRUNC+O_CREAT 700
stderr = RETVAL
map.0=-1
map.1=stdout
param.0='param.0'
'spawn /bin/c89 map.0 param.0 __environment.'
spid = RETVAL
errno = ERRNO
If spid==-1 Then Do
str = 'unnable to spawn param.0', errno='errno
'write' stderr 'str'
Exit_errno
End
'waitpid (spid) waitpid. 0'
xrc = waitpid.W_EXITSTATUS
If xrc<>0 Then Do
str = 'failed, exit status=xrc
'write' stderr 'str'
End
Exit xrc
```

**Running a z/OS UNIX C/C++ application program that is not HFS-resident**

Submit a z/OS UNIX C/C++ application executable program (an executable file that is an MVS PDS or PDSE member) to run under the MVS batch environment using the JCL EXEC statement the same way you would submit a traditional C/C++ application. The POSIX(ON) runtime option should be specified.

**Running COBOL programs under z/OS UNIX**

COBOL programs are supported under z/OS UNIX.

In order to use COBOL under z/OS UNIX, the COBOL programs must be compiled with the Enterprise COBOL for z/OS compiler, COBOL for OS/390 & VM compiler or the COBOL for MVS & VM compiler, and the programs must be compiled with the RENT compiler option.
You can compile and link edit your COBOL programs in the z/OS UNIX shell with the cob2 command. The cob2 command is available with COBOL for OS/390 & VM V2R2 or Enterprise COBOL for z/OS.

Alternatively, you can compile your programs in TSO or batch and have the object module that is written to a z/OS UNIX file by using the PATH parameter instead of the DSNAME parameter for the SYSLIN DD. Once you have your object modules in a z/OS UNIX file, you can use the c89 utility to create an executable file.

When you want to use COBOL programs under z/OS UNIX, be aware of the following situations:

- When COBOL is the main routine of a z/OS UNIX process, parameters are not passed in the C argv and argc format. Instead the parameter list consists of three parameters that are passed by reference:
  1. Argument-count: a binary fullword integer that contains the number of elements in each of the arrays that are passed as the second and third parameters.
  2. Argument-length-list: an array of pointers. The Nth entry in the array is the address of a fullword binary integer that contains the length of the Nth entry in the Argument-list (the third argument).
  3. Argument-list: an array of pointers. The Nth entry in the array is the address of the Nth character string that is passed as an argument on the spawn(), exec(), or command invocation.

- DISPLAY UPON SYSOUT data is written to stdout unless a DD is allocated that matches the value in the OUTDD compiler option.

- In order to run COBOL programs in more than one thread, all of the COBOL programs must be compiled with the Enterprise COBOL compiler using the THREAD compiler option.

- The COBOL MERGE statement and the format 1 SORT statement are not supported. The format 2 SORT statement is supported.

For more information about compiling, link-editing, and running COBOL programs in a z/OS UNIX shell environment, see the appropriate version of the programming guide in the COBOL library at Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733).

**Basic link-editing and running PL/I routines under z/OS UNIX with POSIX(ON)**

**Note:** The following section does not apply to Enterprise PL/I for z/OS; they only apply to the earlier PL/I products.

When the runtime option POSIX(ON) is specified, PL/I routines in the IPT follow the same rules and behave the same as when POSIX(ON) is not in effect.

PL/I routines in non-IPTs, however, must observe the following rules, or the result is unpredictable. No runtime diagnosis is provided to enforce these rules.

- The non-IPT must be created by a C/C++ routine or assembler program. PL/I routines must be reentrant and in AMODE(31). A PL/I routine can be the first routine in the thread. To ensure that the PL/I-specific runtime is available at the time the PL/I routine is running in a non-IPT, one of the following situations must be true in the main executable program:
  - A PL/I routine directly calls a C/C++ routine.
  - A C/C++ routine directly calls a PL/I routine.
  - A PL/I for MVS & VM routine is present in the executable program.
  - Language Environment PL/I signature CSECT CEESG010 is explicitly included in the executable program.

If none of these situations exist in the main executable program and a PL/I routine is going to run in a particular thread, you must do one of the following in that thread:

- Fetch or dynamically call an executable program that contains a PL/I routine.
- Fetch or dynamically call an executable program that contains Language Environment PL/I signature CSECT CEESG010.
• PL/I routines in non-IPTs are supported in the same environments as the C/C++ routines, except the z/OS UNIX shells. The executable form of PL/I routines, however, can run under the shells in conjunction with C/C++ routines in the application using the utilities provided by z/OS UNIX.

• OS PL/I routines can be in non-IPTs. See the appropriate migration guide in the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735) for the OS PL/I executable programs supported under Language Environment.

• The following functions are not supported:
  – PL/I language constructs associated with MTF, the Language Environment PL/I MTF-specific runtime option PLITASKCOUNT, the Language Environment PL/I MTF-specific trace facility via the runtime option TRACE(ON,,LE=2), and the PL/I MTF-specific data set SIBMTASK. The language constructs are:
    - CALL statement with TASK, EVENT, or PRIORITY option
    - EVENT variable
    - COMPLETION and STATUS built-in function and pseudovariable
    - WAIT statement
    - PRIORITY built-in function and pseudovariable
    - DELAY statement
  – I/O using PL/I language statements is not supported except for the limited support provided using the SYSPRINT file and DISPLAY statement.
  – PL/I FETCH and RELEASE statements are not supported.
  – Controlled variables are not supported.
  – Data sharing among threads is limited. Variables must not be referred to across thread boundaries even though the scope of the PL/I names declaration is unchanged.
  – ON-unit inheritance is defined at the thread level. No ON-unit inheritance is provided from the creating threads.

• The following functions are supported with restrictions:
  – SYSPRINT
    If the SYSPRINT file is defined as STREAM OUTPUT EXTERNAL and it is opened in the IPT before any other threads are created, the SYSPRINT file can be shared among the threads. The file must remain open while other threads are using it. The file must be closed explicitly by the IPT or implicitly by Language Environment when the application terminates.
  – DISPLAY
    The DISPLAY statement without the REPLY option and EVENT option is supported.
  – CALL
    The PL/I routine in the non-IPT must not call a subroutine fetched in the initial thread, even if the routine has been fetched before the noninitial thread is created.
  – EXIT
    EXIT is not recommended. When it is used, only the current thread is exited. If the EXIT statement is used in the initial thread, the entire application terminates. There is no defined order as to which thread terminates first.
  – STOP
    STOP is not recommended. When it is used, the entire application terminates. There is no defined order as to which thread terminates first.
  – If a thread is designed to be used many times before it is terminated, reset the user return code by using PLIRETC(0) in the first PL/I routine in the thread.
The z/OS UNIX-defined signals are handled in the same way for PL/I routines in non-IPT and IPT environments. If the z/OS UNIX signals are delivered to PL/I routines, the signals are ignored until the PL/I routine returns.

### Basic link-editing and running PL/I MTF applications under z/OS UNIX

SIBMTASK is provided to create the main executable program for PL/I MTF applications. SIBMTASK must be concatenated before SCEELKED when the main executable program is created, whether you are creating a new PL/I MTF application or relink-editing an existing OS PL/I routine with Language Environment. SIBMTASK replaces PLITASK under OS PL/I. If the main executable program of your application is link-edited with SIBMTASK or OS PL/I PLITASK, but does not use multitasking functions, you might notice some performance loss during initialization and termination. The same releases of OS PL/I executable programs are supported by multitasking as well as nonmultitasking applications. The OS PL/I shared library support is the same for both multitasking and nonmultitasking applications.

PL/I multitasking follows the Language Environment program management model discussed in Chapter 13, “Program management model,” on page 139. PL/I MTF supports a single Language Environment process within an address space, and is supported in the initial enclave only. If a PL/I multitasking application contains nested enclaves, the initial enclave must contain a single task. Violation of any of these rules is not diagnosed and is likely to cause unpredictable results.

The POSIX(ON) runtime option is not supported for a PL/I MTF application and therefore no programs, including assembler programs, in the application can invoke any POSIX functions. If POSIX(ON) is in effect when a multitasking main executable program is encountered, the application will abend.

For a more detailed discussion of PL/I MTF support, see the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735). For more information about migration considerations for OS PL/I MTF applications, see the appropriate migration guide in the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).
Chapter 8. Using IBM-supplied cataloged procedures

A cataloged procedure is a set of job control statements that are stored in a system library (for example SYS1.PROCLIB). The storage location for cataloged procedures is installation-defined and might differ at your location from what is shown here.

Within a cataloged procedure, there are typically one or more EXEC statements, each of which can be followed by one or more DD statements. You can retrieve a cataloged procedure from the library by using its member name in an EXEC statement of a job control statement in the input stream.

Cataloged procedures can contain statements for the processing of an entire job, or statements to process one or more steps of a job, with the remaining steps defined in job control statements in the input stream. A job can use several cataloged procedures to process one or more of the job steps, or it can use the same cataloged procedure in more than one job step.

You can use cataloged procedures to save time and reduce JCL errors. If the statements in the procedure do not match your requirements exactly, you can easily modify them or add new statements for the duration of a job.

The cataloged procedures shown in this section are intended for use as references and do not necessarily reflect the procedures as they are provided at your installation. If options are not explicitly supplied with the procedure, default options established at the installation apply. You can override these default options by using an EXEC statement that includes the desired options (see “Overriding and adding to EXEC statements” on page 97).

Invoking cataloged procedures

To invoke a cataloged procedure, specify its name in the PROC parameter of an EXEC statement. You do not need to code the keyword PROC. For example, to use the cataloged procedure CEEWLG, include the following statement in an appropriate position among your other job control statements in the input stream:

//stepname EXEC PROC=CEEWLG
or
//stepname EXEC CEEWLG

Either of these EXEC statements can be used to call the IBM-supplied cataloged procedure CEEWLG to process the job step specified in stepname.

A job that calls for a cataloged procedure to run can also contain DD statements that are applicable to the cataloged procedure, such as:

- Other cataloged procedures to be run
- Other (single or multiple) executions of the same cataloged procedure
- Executable programs to be run

Step names in cataloged procedures

The stepname in a cataloged procedure is the same as the abbreviated processor name. For example, the step that executes a compiled and link-edited program is named GO. In the procedure named CEEWLG (see “CEEWLG — Link and run a Language Environment conforming non-XPLINK program” on page 92), the first step is named LKED, and the second is named GO. Some of the PROCs provided are for creating constructed reentrant C/C++ executables. See “Making your C/C++ program reentrant” on page 121 for a description of constructed reentrant C/C++ programs.
Some of these PROCs use a prelink step, for when the prelinker must still be used. Some have a P in their names, such as EDCPL, to denote the prelink step PLKED. Other PROCs, for which the prelinker was mandatory when the linkage editor was used (such as CBCCL) have just an L in their names, to denote both the prelink step PLKED and the link-edit step LKED.

PROCs which must use the binder (that is they cannot use the linkage editor), have a B in their names to denote the binder step BIND, such as CBCB. These typically have counterpart PROCs with an L in their names, such as CBCL, which can be used when the linkage editor must be used. These may also use the prelinker, and therefore may or may not have a P in their names.

PROCs which have an L in their names to denote the link-edit step LKED, which do not have a counterpart PROC with a B in their names, do not have a prelink step. These generic link-edit PROCs can be used with the binder, and also will work correctly with the linkage editor. In order for these generic link-edit PROCs to work with constructed reentrant C/C++ programs, the appropriate overrides must be used. See “Modifying cataloged procedures” on page 97 for some examples.

### Unit names in cataloged procedures

The esoteric unit name used in IBM-supplied cataloged procedures is one of the following:

- UNIT=SYSDA
- UNIT=VIO

Esoteric unit names are defined during system initialization and installation; the installation should maintain a list of esoteric unit names. Both these names may be defined as VIO (virtual I/O) data sets. See z/OS MVS JCL User’s Guide for more information about VIO data sets.

All of the data sets that can be created use one of these esoteric units names. Most of these are set up as temporary data sets, and some are typically overridden to become permanent data sets, by using procedure parameters, or by overriding procedure statements. See “Modifying cataloged procedures” on page 97 for more information about overriding statements in cataloged procedures.

### Data set names in cataloged procedures

When you use DSNAME=&amp;name in a DD statement, it is a temporary data set that is deleted when the job terminates. If you want the data set to be kept, override the DD statement with a permanent data set name and specify the appropriate DISP parameters.

See “Required DD Statements” under “Writing JCL for the link-edit process” on page 60 for a detailed description of each of the data sets included in the cataloged procedures discussed in this topic. See “Overriding and adding to EXEC statements” on page 97 for instructions about overriding DD statements in cataloged procedures.

### IBM-supplied cataloged procedures

The IBM-supplied cataloged procedures that you can use are listed in Table 15 on page 88.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Name</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load and run a Language Environment-conforming non-XPLINK program</td>
<td>CEEWG</td>
<td>“CEEWG — Load and run a Language Environment conforming non XPLINK program” on page 91</td>
</tr>
<tr>
<td>Link-edit a Language Environment-conforming non-XPLINK program</td>
<td>CEEWL</td>
<td>“CEEWL — Link a Language Environment conforming non XPLINK program” on page 92</td>
</tr>
<tr>
<td>Procedure</td>
<td>Name</td>
<td>For more information, see:</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Link-edit and run a Language Environment-conforming non-XPLINK program</td>
<td>CEEWLG</td>
<td>“CEEWLG — Link and run a Language Environment conforming non-XPLINK program” on page 92</td>
</tr>
<tr>
<td>Load and run a Language Environment-conforming XPLINK program</td>
<td>CEEXR</td>
<td>“CEEXR — Load and run a Language Environment conforming XPLINK program” on page 93</td>
</tr>
<tr>
<td>Link-edit a Language Environment-conforming XPLINK program</td>
<td>CEEXL</td>
<td>“CEEXL — Link-edit a Language Environment conforming XPLINK program” on page 93</td>
</tr>
<tr>
<td>Link-edit and run a Language Environment-conforming XPLINK program</td>
<td>CEEXLR</td>
<td>“CEEXLR — Link and run a Language Environment conforming XPLINK program” on page 94</td>
</tr>
<tr>
<td>Compile a C program</td>
<td>EDCC</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile and link-edit a C program</td>
<td>EDCCL</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile and bind a C program</td>
<td>EDCCB</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile, bind, and run a C program</td>
<td>EDCCBG</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile, link-edit, and run a C program</td>
<td>EDCCLG</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile, prelink, link-edit, and run a C program</td>
<td>EDCGPLG</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Prelink and link-edit a C program</td>
<td>EDCPL</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile a C++ program</td>
<td>CBCC</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile and bind a C++ program</td>
<td>CBCCB</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile, bind, and run a C++ program</td>
<td>CBCCBG</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Bind a C++ program</td>
<td>CBCB</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Bind and run a C++ program</td>
<td>CBCBG</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile, prelink, and link-edit a C++ program</td>
<td>CBCCL</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Compile, prelink, link-edit, and run a C++ program</td>
<td>CBCCLG</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Prelink and link-edit a C++ program</td>
<td>CBCL</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Prelink, link-edit, and run a C++ program</td>
<td>CBCLG</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Run a C++ program</td>
<td>CBCG</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Invoke the <code>iconv</code> (character-conversion) utility</td>
<td>EDCICONV</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Invoke the <code>genxlt</code> (generate a translate table) utility</td>
<td>EDCGNXLT</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Invoke the DSECT conversion utility</td>
<td>EDCDSECT</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Invoke the locale object utility</td>
<td>EDCLDEF</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
<tr>
<td>Invoke the maintain an object library utility</td>
<td>EDCLIB</td>
<td>z/OS XL C/C++ User’s Guide</td>
</tr>
</tbody>
</table>
Table 15: IBM-supplied cataloged procedures (continued)

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Name</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoke the compile and maintain an object</td>
<td>EDCCLIB</td>
<td><em>z/OS XL C/C++ User's Guide</em></td>
</tr>
<tr>
<td>library utility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invoke the demangle mangled names utility</td>
<td>CXXFILT</td>
<td><em>z/OS XL C/C++ User's Guide</em></td>
</tr>
<tr>
<td>Compile a COBOL program</td>
<td>IGYWC</td>
<td>The appropriate version of the COBOL programming guide in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COBOL library at Enterprise COBOL for z/OS library (<a href="http://www.ibm.com/">www.ibm.com/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>support/docview.wss?uid=swg27036733).</td>
</tr>
<tr>
<td>Compile and link-edit a COBOL program</td>
<td>IGYWCL</td>
<td>The appropriate version of the COBOL programming guide in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COBOL library at Enterprise COBOL for z/OS library (<a href="http://www.ibm.com/">www.ibm.com/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>support/docview.wss?uid=swg27036733).</td>
</tr>
<tr>
<td>Compile, link-edit, and run a COBOL program</td>
<td>IGYWCLG</td>
<td>The appropriate version of the COBOL programming guide in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COBOL library at Enterprise COBOL for z/OS library (<a href="http://www.ibm.com/">www.ibm.com/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>support/docview.wss?uid=swg27036733).</td>
</tr>
<tr>
<td>Compile, prelink, and link-edit a COBOL program</td>
<td>IGYWCPL</td>
<td>The appropriate version of the COBOL programming guide in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COBOL library at Enterprise COBOL for z/OS library (<a href="http://www.ibm.com/">www.ibm.com/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>support/docview.wss?uid=swg27036733).</td>
</tr>
<tr>
<td>Compile, prelink, link-edit, and run a COBOL</td>
<td>IGYWPLG</td>
<td>The appropriate version of the COBOL programming guide in the</td>
</tr>
<tr>
<td>program</td>
<td></td>
<td>COBOL library at Enterprise COBOL for z/OS library (<a href="http://www.ibm.com/">www.ibm.com/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>support/docview.wss?uid=swg27036733).</td>
</tr>
<tr>
<td>Compile, load, and run a COBOL program</td>
<td>IGYWC</td>
<td>The appropriate version of the COBOL programming guide in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COBOL library at Enterprise COBOL for z/OS library (<a href="http://www.ibm.com/">www.ibm.com/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>support/docview.wss?uid=swg27036733).</td>
</tr>
<tr>
<td>Prelink and link-edit a COBOL program</td>
<td>IGYWPL</td>
<td>The appropriate version of the COBOL programming guide in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>COBOL library at Enterprise COBOL for z/OS library (<a href="http://www.ibm.com/">www.ibm.com/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>support/docview.wss?uid=swg27036733).</td>
</tr>
<tr>
<td>Link-edit a Fortran program</td>
<td>AFHWL</td>
<td>“AFHWL — Link a program written in Fortran” on page 95</td>
</tr>
<tr>
<td>Link-edit and run a Fortran program</td>
<td>AFHWLGE</td>
<td>“AFHWLGE — Link and run a program written in Fortran” on page 96</td>
</tr>
<tr>
<td>Change any external names in conflict</td>
<td>AFHWN</td>
<td>“AFHWN — Resolving name conflicts between C and Fortran” on page</td>
</tr>
<tr>
<td>between C and Fortran to the Fortran-</td>
<td></td>
<td>96</td>
</tr>
<tr>
<td>recognized name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following sections provide more details about, and example invocations of, the language-independent cataloged procedures CEEWG, CEEWL, CEEWLG, CEEXR, CEEXL and CEEEXLR.

**CEEWG — Load and run a Language Environment conforming non XPLINK program**

The CEEWG cataloged procedure that is shown in Figure 36 on page 92 includes the GO step, which loads an object module produced by the compiler and executes the load module.

The following DD statement, indicating the location of the object module, must be supplied in the input stream to the GO step:

```plaintext
//GO.SYSIN DD * (or appropriate parameters)
```

The data set SCEELKED must be included in your link-edit SYSLIB concatenation. This is the name of the Language Environment resident library. (The high-level qualifier of this resident library might have been changed at your installation.)

The data set SCEERUN must be included in the STEPLIB DD statement for the GO step. (The name of this load library might have been changed at your installation.)

If the application refers to any data sets in the execution step (such as user-defined files or SYSIN), DD statements that define these data sets must be provided.
IBM-supplied cataloged procedures

//CEEWG    PROC LIBPRFX='CEE'
//GO       EXEC PGM=LOADER,REGION=2048K
//SYSLIB   DD   DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSLOUT  DD   SYSOUT=*  
//SYSLINS  DD   DSNAME=&SIN   
//SYSLIN   DD   DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD   SYSOUT=* 
//CEEDUMP  DD   SYSOUT=* 
//SYSUDUMP DD   SYSOUT=* 

Figure 36: Cataloged procedure CEEWG, which loads and runs a program written in any Language
Environment-conforming HLL

Note: CEEWG does not work for program objects with deferred load classes, such as executables that are
produced by COBOL V5 and V6.

CEEWL — Link a Language Environment conforming non XPLINK program

The CEEWL cataloged procedure shown in Figure 37 on page 92 includes the LKED step that invokes the
binder (symbolic name HEWL) to link edit an object module.

The following DD statement, indicating the location of the object module, must be supplied in the input
stream:

//LKED.SYSIN DD *      (or appropriate parameters)

The data set SCEELKED must be included in your link-edit SYSLIB concatenation. This is the name of the
Language Environment link-edit library. (The high-level qualifier of this link-edit library might have been
changed at your installation.)

//CEEWL    PROC LIBPRFX='CEE',
//          PGMLIB='&&GOSET',GOPGM=GO
//LKED     EXEC PGM=HEWL,REGION=1024K
//SYSLIB   DD   DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSLOUT  DD   SYSOUT=*  
//SYSLINS  DD   DSNAME=&SYSIN   
//SYSLIN   DD   DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD   SYSOUT=* 
//CEEDUMP  DD   SYSOUT=* 
//SYSDUMP  DD   SYSOUT=* 

Figure 37: Cataloged procedure CEEWL, which link-edits a program written in any Language Environment-
conforming HLL

CEEWLG — Link and run a Language Environment conforming non-XPLINK program

The CEEWLG cataloged procedure in Figure 38 on page 93 includes the LKED step, which invokes the
binder (symbolic name HEWL) to link-edit an object module, and the GO step, which executes the
executable program produced in the first step.

The following DD statement, indicating the location of the object module, must be supplied in the input
stream:

//LKED.SYSIN DD *      (or appropriate parameters)

The data set SCEELKED must be included in your link-edit SYSLIB concatenation. This is the name of the
Language Environment link-edit library. (The high-level qualifier of this link-edit library might have been
changed at your installation.)
The data set SCEERUN must be included in the STEPLIB DD statement for the GO step. (The name of this load library might have been changed at your installation.)

If the application refers to any data sets in the execution step (such as user-defined files or SYSIN), you must also provide DD statements that define these data sets.

```
//CEEWLG   PROC LIBPRFX='CEE',GOPGM=GO
//LKED    EXEC PGM=HEWL,REGION=1024K
//SYSLIB   DD   DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD   SYSOUT=* 
//SYSLIN   DD   DDNAME=SYSIN
//SYSLMOD  DD   DSNAME=&GOSET(&GOPGM),SPACE=(TRK,(10,10,1)),
              UNIT=SYSDA,DISP=(MOD,PASS)
//SYSLMOD  DD   DSNAME=&PGMLIB,DISP=SHR
//GO       EXEC PGM=*.LKED.SYSLMOD,COND=(4,LT,LKED),REGION=2048K
//STEPLIB  DD   DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD   SYSOUT=* 
//CEEDUMP  DD   SYSOUT=* 
//SYSDUMP  DD   SYSOUT=* 
```  

Figure 38: Cataloged procedure CEEWLG, which link-edits and runs a program written in any Language Environment-conforming HLL

### CEEXR — Load and run a Language Environment conforming XPLINK program

The CEEXR cataloged procedure shown in Figure 39 on page 93 includes the GO step, which loads and executes an XPLINK program module specified on input parameters to the procedure.

The data sets SCEERUN and SCEERUN2 must be included in the STEPLIB DD statement for the GO step. (The high-level qualifier of these load libraries might have been changed at your installation.)

If the application refers to any data sets in the execution step (such as user-defined files), DD statements that define these data sets must be provided.

```
//CEEXR    PROC PGMLIB=,                < INPUT STEPLIB ... REQUIRED
//   GOPGM=,                            < INPUT PROGRAM ... REQUIRED
//   LIBPRFX='CEE',                     < PREFIX FOR LIBRARY DSN
//   GREGSIZ='2048K',                   < EXECUTION REGION SIZE
// GPARMS='XPLINK(ON),TERMTHDACT(UADUMP)/' < RUN-TIME OPTS
//GO       EXEC PGM=&GOPGM,REGION=&GREGSIZ,
//         PARM='&GPARMS'
//STEPLIB  DD   DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//          DD   DSNAME=&LIBPRFX..SCEERUN2,DISP=SHR
//          DD   DSNAME=&PGMLIB,DISP=SHR
//SYSPRINT DD   SYSOUT=* 
//CEEDUMP  DD   SYSOUT=* 
//SYSDUMP  DD   SYSOUT=* 
```  

Figure 39: Cataloged procedure CEEXR, which loads and runs a program-compiled XPLINK

### CEEXL — Link-edit a Language Environment conforming XPLINK program

The CEEXL cataloged procedure shown in Figure 40 on page 94 includes the LKED step that invokes the Binder (symbolic name IEWL) to link-edit an object module specified on input parameters to the procedure.

Any sidedecks that are needed to resolve references in this object module to DLLs must be specified on a SYSIMP DD statement.
The data set SCEEBIND must be included in your link-edit SYSLIB concatenation. This is the name of the Language Environment link-edit library for XPLINK applications. (The high-level qualifier of this link-edit library might have been changed at your installation.)

```
//CEEXL   PROC INFILE=,                < INPUT ... REQUIRED
//   LIBPRFX='CEE',                     < PREFIX FOR LIBRARY DSN
//   LREGSIZ='20M',                     < BINDER REGION SIZE
//   LPARMS='MAP,LIST=NOIMP',           < ADDITIONAL BINDER PARMS
//   OUTFILE='&GSET(GO),DISP=(NEW,PASS),UNIT=SYSALLDA,SPACE=(TRK,(7,7,
//           1)),DSNTYPE=LIBRARY'
//LKED     EXEC PGM=IEWL,REGION=&LREGSIZ,
//   PARM='AMODE=31,RENT,DYNAM=DLL,CASE=MIXED,&LPARMS'
//SYSLIB   DD   DSNAME=&LIBPRFX..SCEEBIND,DISP=SHR
//SYSLIN   DD   DSNAME=&LIBPRFX..SCEEBIND,DISP=SHR
//SYSPRINT DD   SYSOUT=*  
//SYSLMOD  DD   DSNAME=&OUTFILE
//SYSUT1   DD   UNIT=SYSALLDA,SPACE=(TRK,(10,10))
//SYSPFS   DD   DSNAME=&OUTFILE
//SYSin    DD   DUMMY
```

Figure 40: Cataloged procedure CEEXL, which link-edits a program-compiled XPLINK

CEEXLR — Link and run a Language Environment conforming XPLINK program

The CEEXLR cataloged procedure shown in Figure 41 on page 95 includes the LKED step, which invokes the Binder (symbolic name IEWL) to link-edit an object module, and the GO step, which executes the program module produced in the first step.

Any sidedecks that are needed to resolve references in this object module to DLLs must be specified on a SYSIMP DD statement.

The data set SCEEBIND must be included in your link-edit SYSLIB concatenation. This is the name of the Language Environment link-edit library for XPLINK applications. (The high-level qualifier of this link-edit library might have been changed at your installation.)

The data sets SCEERUN and SCEERUN2 must be included in the STEPLIB DD statement for the GO step. (The high-level qualifier of these load libraries might have been changed at your installation.)
If the application refers to any data sets in the execution step (such as user-defined files or SYSIN), DD statements that define these data sets must be provided.

```plaintext
//CEEXLR   PROC INFILE=*,            < INPUT ... REQUIRED
//   LIBPRFX='CEE',                     < PREFIX FOR LIBRARY DSN
//   LREGSIZ='20M',                     < BINDER REGION SIZE
//   LPARMS='MAP,LIST=NOIMP',           < ADDITIONAL BINDER PARMS
//   GREGSIZ='2048K',                   < EXECUTION REGION SIZE
//   GPARMs='XPLINK(ON),TERMTHDACT(UADUMP)/', < RUN-TIME OPTS
//   OUTFILE='&&GSET(GO),DISP=(NEW,PASS),UNIT=SYSALLDA,SPACE=(TRK,(7,7,
//   1)),DSNTYPE=LIBRARY'
//LKED     EXEC PGM=IEWL,REGION=&LREGSIZ,
//         PARM='AMODE=31,RENT,DYNAM=DLL,CASE=MIXED,&LPARMS'
//SYSLIB   DD   DSNAME=&LIBPRFX..SCEEBIND,DISP=SHR
//SYSPRINT DD   SYSOUT=*               < OUTPUT ... REQUIRED
//SYSLIN   DD   DSNAME=&INFILE,DISP=SHR
//          DD   DSNAME=&LIBPRFX..SCEELIB(CELHS003),DISP=SHR
//          DD   DSNAME=&LIBPRFX..SCEELIB(CELHS001),DISP=SHR
//          DD   DDNAME=SYSIN
//SYSLMOD  DD   DSNAME=&OUTFILE
//SYSU1    DD   UNIT=SYSALLDA,SPACE=(TRK,(10,10))
//SYSDEFSD DD   DUMMY
//GO       EXEC PGM=*.LKED.SYSLMOD,COND=(4,LT,LKED),REGION=&GREGSIZ,
//         PARM='&GPARMS'
//STEPLIB  DD   DSNAME=&LIBPRFX..SAFHFORT,DISP=SHR
//         DD   DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//         DD   DDNAME=SYSIN
//SYSDUMP DD   DUMMY
//SYSMOD   DD   DUMMY
//SYSSYSIN DD   DUMMY
```

Figure 41: Cataloged procedure CEEXLR, which link-edits and runs a program-compiled XPLINK

**AFHWL — Link a program written in Fortran**

The AFHWL cataloged procedure shown in Figure 42 on page 95 includes the LKED step, which invokes the binder (symbolic name HEWL) to link-edit an object module. The procedure can be used to link-edit applications containing Fortran or assembler routines having names that conflict with existing C library routines, as discussed in "Resolving library module name conflicts between Fortran and C" on page 15.

The following DD statement, indicating the location of the object module, must be supplied in the input stream:

```plaintext
//LKED.SYSIN DD *    (or appropriate parameters)
```

The data sets SAFHFOR and SCEELKED must both be included in your link-edit SYSLIB concatenation (in that order). SAFHFOR is the name of the Fortran-specific link-edit library and is used to resolve certain Fortran intrinsic function names. SCEELKED is the name of the Language Environment link-edit library. (The high-level qualifier of this link-edit library might have been changed at your installation.)

```plaintext
//AFHWL   PROC LIBPRFX='CEE',
//          PGMLIB='&&GOSET',GOPGM=GO
//LKED     EXEC PGM=HEWL,REGION=1024K
//SYSLIB   DD   DSNAME=&LIBPRFX..SAFHFORT,DISP=SHR
//          DD   DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD   SYSOUT=*               < OUTPUT ... REQUIRED
//SYSLMOD  DD   DDNAME=SYSIN
//SYSSYSIN DD   DUMMY
```

Figure 42: Using AFHWL to link a program written in Fortran
AFHWLG — Link and run a program written in Fortran

The AFHWLG cataloged procedure shown in Figure 43 on page 96 includes the LKED step, which invokes the binder (symbolic name HEWL) to link-edit an object module, and the GO step, which executes the executable program produced in the first step. The procedure can be used to link-edit and run applications containing Fortran or assembler routines having names that conflict with existing C library routines, as discussed in “Resolving library module name conflicts between Fortran and C” on page 15.

The following DD statement, indicating the location of the object module, must be supplied in the input stream:

```
//LKED.SYSIN DD *       (or appropriate parameters)
```

The data sets SAFHFORT and SCEELKED must both be included in your link-edit SYSLIB concatenation (in that order). SAFHFORT is the name of the Fortran-specific link-edit library and is used to resolve certain Fortran intrinsic function names. SCEELKED is the name of the Language Environment link-edit library. (The high-level qualifier of this link-edit library might have been changed at your installation.)

The data set SCEERUN must also be included in the STEPLIB DD statement for the GO step. (The name of the load library might have been changed at your installation.)

If the application refers to any data sets in the execution step (such as user-defined files or SYSIN), you must also provide DD statements that define these data sets.

```
//AFHWLG   PROC LIBPRFX='CEE'
//LKED     EXEC PGM=HEWL,REGION=1024K
//SYSLIB   DD   DSNAME=&LIBPRFX..SAFHFORT,DISP=SHR
//         DD   DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD   SYSOUT=*   
//SYSLIN   DD   DDNAME=SYSIN 
//SYSLMOD  DD   DSNAME=&GOSET(GO),
//             SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS) 
//SYST1    DD   UNIT=SYSDA,SPACE=(TRK,(10,10))
//GO       EXEC PGM=*.LKED.SYSLMOD,COND=(4,LT,LKED),REGION=2048K
//STEPLIB  DD   DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD   SYSOUT=* 
//CEEDUMP   DD   SYSOUT=* 
//SYSDUMP   DD   SYSOUT=* 
```

Figure 43: Procedure AFHWLG, used to link and run a program written in Fortran

AFHWN — Resolving name conflicts between C and Fortran

The AFHWN cataloged procedure shown in Figure 44 on page 97 includes the LKED step, which invokes the binder (symbolic name HEWL) to link-edit an object module.

The following DD statement, indicating the location of the object module, must be supplied in the input stream:

```
//LKED.SYSIN DD *       (or appropriate parameters)
```

The contents of the SYSIN data set must include the CHANGE statements in member AFHWNCH in SCEESAMP and the object module itself. See “Resolving library module name conflicts between Fortran and C” on page 15 for further details.

The data sets SAFHFORT and SCEELKED must both be included in your link-edit SYSLIB concatenation (in that order). SAFHFORT is the name of the Fortran-specific link-edit library and is used to resolve certain Fortran intrinsic function names. SCEELKED is the name of the Language Environment link-edit library. The high-level qualifier of this link-edit library might have been changed at your installation. Figure 44 on page 97 shows the LKED in AFHWN.
### Modifying cataloged procedures

You can modify the statements of a cataloged procedure for the duration of the job step in which it is invoked, either by overriding one or more parameters in the EXEC or DD statements or by adding DD statements to the procedure. Any parameter in a cataloged procedure, except the PGM=programe parameter in the EXEC statement, can be overridden. Parameters or statements not specified in the procedure can also be added. When a cataloged procedure is overridden or added to, the changes apply only during one execution. The changes do not affect the master copy of the cataloged procedure stored in the procedure library.

The following sections discuss overriding and adding to EXEC and DD statements, respectively. For complete details, see [z/OS MVS JCL Reference](#).

#### Overriding and adding to EXEC statements

A parameter with a qualified name (qualified by the procedure step in which it is specified) applies only to the EXEC statement in which it is specified. If a parameter of an EXEC statement that invokes a cataloged procedure has an unqualified name, the parameter applies to all the EXEC statements in the cataloged procedure. For example, REGION=2048 specifies a region size of 2048 for all of the EXEC statements in a given procedure, whereas REGION.GO=2048 applies to only the GO step of the procedure.

If you want to modify a multiple step procedure, you can do so by specifying parameters with qualified names on a step-by-step basis. If you want to modify the entire procedure, specify the name of the parameter in an EXEC statement without qualifying it. The modifications override existing parameters in the cataloged procedure.

#### Overriding and adding DD statements

You can override or add a DD statement by specifying a DD statement whose name is composed of the ddname of the DD statement being overridden, preceded by the procedure stepname that qualifies that ddname:

```plaintext
//procstep.ddname DD (appropriate parms)
```

You must observe the following when overriding or adding a DD statement within a step in a procedure:

- Overriding DD statements must be in the same order in the input stream as they are in the cataloged procedure.
- DD statements to be added must follow overriding DD statements.

Additionally, you should be aware of the following when overriding a DD statement:

---

Figure 44: Cataloged procedure AFHWN, used in resolving name conflicts

```plaintext
//AFHWN    PROC LIBPRFX='CEE',
           PGMLIB='&&GOSET',GOPGM=GO
//LKED     EXEC PGM=HEWL,REGION=1024K,PARM='NCAL,LET'
//SYSLIB   DD   DSNAME=&LIBPRFX..SAFHFORT,DISP=SHR
//         DD   DSNAME=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD   SYSOUT=*  
//SCEESAMP DD   DSNAME=&LIBPRFX..SCEESAMP,DISP=SHR
//SYSLIN   DD   DSNAME=SYSIN
//SYSLOLD  DD   DSNAME=&&PGMLIB(&GOPGM),
//            SPACE=(TRK,(10,10,1)),
//            UNIT=SYSDA,DISP=(MOD, PASS)
//SYSUT1   DD   UNIT=SYSDA,SPACE=(TRK,(10,10))
```
• To nullify a keyword parameter (except the DCB and AMP parameters), write the keyword and an equal sign followed by a comma in the overriding DD statement. For example, to nullify the use of the UNIT parameter, specify UNIT=, in the overriding DD statement.

• You can nullify a parameter by specifying a mutually exclusive parameter. For example, you can nullify the SPACE parameter by specifying the mutually exclusive SPLIT parameter in the overriding DD statement.

• There is no order of precedence for the parameters. Their placement (order of execution) does not matter.

• To override DD statements in a concatenation of data sets, you must provide one DD statement for each data set in the concatenation. Only the first DD statement in the concatenation should be named. If the DD statement you want to change or add follows one or more DD statements that will be unchanged, code one DD statement with blank operand for each unchanged DD statement ahead of the first DD statement that you want to change or add.

For example, to add your load module data set, MY.LIB, to the runtime STEPLIB in a Language Environment cataloged procedure containing one data set whose DD statement you do not want to change, code:

```
//GO.STEPLIB DD
// DD DSN=MY.LIB,DISP=SHR
```

This causes your load module data set to be searched after the data set named in the STEPLIB DD statement in the cataloged procedure.

To have another data set searched before any data sets already in the cataloged procedure, you must specify all the data sets in your overriding DD statements. For example, to have CEE.SIBMMATH searched before CEE.SCEELKED (assuming it is a non-XPLINK application), code:

```
//LKED.SYSLIB DD DSN=CEE.SIBMMATH,DISP=SHR
// DD DSN=CEE.SCEELKED,DISP=SHR
```

This causes the PL/I versions of the math routines to be link-edited into your load module rather than the identically named SCEELKED math routines.

• If the DDNAME=ddname parameter is specified in a cataloged procedure, it cannot be overridden; rather, it can refer to a DD statement supplied at the time of execution.

The example in Figure 45 on page 98 shows how to override parameters in a cataloged procedure by:

• Changing the library prefix for the SCEELKED link library to SYS1
• Increasing the region for linking and running the application
• Passing the RPTSTG and RPTOPTS options to the load module when it is executed in the GO step of the procedure
• Specifying PROGRAM1 in USER.OBJLIB as the input object module to the binder

```
//CEEWLG JOB
//*
//LINKGO EXEC CEEWLG,
// LIBPRFX='SYS1',
// REGION=2048K,
// PARM.GO='RPTSTG(ON) RPTOPTS(ON)/'
//*
//LKED.SYSIN DD DSN=USER.OBJLIB(PROGRAM1),DISP=SHR
//*
```

Figure 45: Overriding parameters in the CEEWLG cataloged procedure
Overiding generic link-edit procedures for constructed reentrant programs

To use generic link-edit procedures (as described in “Step names in cataloged procedures” on page 87), both the EXEC statement parameters and DD statements may need to be overridden.

The following are some examples of how to invoke the CEEWL PROC.

1. Creating a C executable which may be constructed reentrant:
   - Parameter COMPAT(CURRENT) assures the highest level of program object will be produced.
   - SCEELKEX allows direct resolution of C/C++ language function names.
   - DSNTYPE=LIBRARY assures that the output data set will be a PDSE (rather than a PDS). STORCLAS may also have to be specified for a new SMS managed data set.
   - Object module USER.OBJ(PROGRAM1) is input.

   ```
   //CEEWL JOB
   //*
   //SETLIB SET LIBPRFX=CEE
   //SETUSER SET USER=USER1
   //*
   //LINK EXEC CEEWL,
   // LIBPRFX=LIBPRFX.,
   // PARM.LKED='COMPAT(CURRENT)'
   //LKED.SYSLIB DD DSN=LIBPRFX..SCEELKEX,DISP=SHR
   // LKED.SYSLMOD DD DSNTYPE=LIBRARY
   //LKED.SYSIN DD DSN=USER.OBJ(PROGRAM1),DISP=SHR
   //*
   ```

2. Create a C DLL, which can also be used as an autocall library. In addition to the previous example:
   - DYNAM(DLL) causes the import and export information to be created and stored in the executable.
   - ALIASES(ALL) causes hidden aliases to be created for all external functions and variables, for subsequent use as an autocall library.
   - Program object USER.LOADLIB(CDLL) is output.
   - Definition sidedeck USER.EXP(CDLL) is output (it contains IMPORT statements for all exported symbols).

   ```
   //CEEWL JOB
   //*
   //SETLIB SET LIBPRFX=CEE
   //SETUSER SET USER=USER1
   //*
   //LINK EXEC CEEWL,
   // PARM.LKED='COMPAT(CURR),DYNAM(DLL),ALIASES(ALL)'
   //LKED.SYSLIB DD DSN=LIBPRFX..SCEELKEX,DISP=SHR
   // LKED.SYSLMOD DD DSN=USER.LOADLIB(CDLL),DISP=SHR,
   // DSNTYPE=LIBRARY
   //LKED.SYSDEFSD DD DSN=USER.EXP(CDLL),DISP=SHR
   //*
   //LKED.SYSIN DD DSN=USER.OBJ(PROGRAM1),DISP=SHR
   //*
   ```
IBM-supplied cataloged procedures
Chapter 9. Using runtime options

This topic describes Language Environment runtime option specification methods and runtime compatibility considerations.

Language Environment provides a set of IBM-supplied default runtime options that control certain aspects of program processing. A system programmer can modify the IBM-supplied defaults on a system-level or region-level basis to suit most applications at their site. An application programmer can further refine these options for individual programs. When an application runs, runtime options are merged in a specific order of precedence to determine the actual values in effect. For more information, see “Order of precedence” on page 103.

For syntax and detailed information about individual runtime options, including how Language Environment runtime options map to specific HLL options, see z/OS Language Environment Programming Reference.

Methods available for specifying runtime options

Language Environment runtime options can be specified in the following ways:

As system-level defaults
Runtime options can be established as system-level defaults 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 a SET CEE=xx command. Individual options can be changed with a SETCEE command.

For more information about specifying system-level default options, see z/OS MVS Initialization and Tuning Reference and z/OS Language Environment Customization.

As region-level defaults
The CEEXOPT macro can be used to create a CEEROPT load module to establish defaults for a particular region. CEEROPT is optional, but if it is used, code just the runtime options to be changed. Runtime options that are omitted from CEEROPT will remain the same as the system-level defaults (if present) or IBM-supplied defaults. The CEEROPT module resides in a user-specified load library.

For more information about specifying region-level defaults, see z/OS Language Environment Customization.

In the CLER CICS transaction
The CICS transaction CLER allows you to display all the current Language Environment runtime options for a region, and to also to modify a subset of these options.

The following runtime options can be modified with the CLER transaction:

- ALL31(ON|OFF)
- CBLPSHPOP(ON|OFF)
- CHECK(ON|OFF)
- HEAPZONES(0-1024,QUIET|MSG|TRACE|ABEND)
- INFOMSGFILTER(ON|OFF)
- RPTOPTS(ON|OFF)
- RPTSTG(ON|OFF)
- TERMTHDACT(QUIET|MSG|TRACE|DUMP|UAONLY|UATRACE|UADUMP|UAIMM)
- TRAP(ON|OFF)
Setting RPTOPTS(ON) or RPTSTG(ON) in a production environment can significantly degrade performance. Also, if ALL31(OFF) is set in a production environment, the stack location will be set to BELOW the 16 MB line, which could cause the CICS region to abend due to lack of storage.

The LAST WHERE SET column of the Language Environment runtime options report contains CICS CLER Trans for those options that were set by CLER.

**Note:** CICS TS 3.1 and higher supports XPLINK programs in a CICS environment. The CLER transaction does not affect the runtime options for these programs.

### As application defaults
The CEEUOPT assembler language source program sets application defaults using the CEEXOPT macro. The CEEUOPT source program can be edited and assembled to create an object module, CEEUOPT. The CEEUOPT object module must be linked with an application to establish application defaults.

### In the assembler user exit
See “CEEBXITA assembler user exit interface” on page 378 for information about how to specify a list of runtime options in the assembler user exit.

For IMS, the UPDATE LE command can be used with the IMS-supplied CEEBXITA exit, DFSBXITA, to allow dynamic overrides for runtime options. For more information about the UPDATE LE command and the IMS-supplied CEEBXITA exit, DFSBXITA, see the IBM Knowledge Center at IMS in IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSEPH2).

### In the storage tuning user exit
The storage tuning user exit can be used to set the Language Environment storage options STACK, LIBSTACK, HEAP, ANYHEAP, and BELOWHEAP. See z/OS Language Environment Customization for information about the storage tuning user exit.

**Note:** Vendor Heap Manager activity is not handled by the Language Environment storage tuning user exit.

### In TSO/E commands, on application invocation
You can specify runtime options as options on the CALL command. See “Running your application under TSO/E” on page 78 for more information.

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

### As 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 59 and “Specifying runtime options with the CEEOPTS DD card” on page 58 for details.

### In your source code:
#### C and C++
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 `pragma` can precede `#pragma runopts`.

Specify `#pragma runopts` as follows:

```
#pragma runopts (option)
```
where option is a Language Environment runtime option.

For more information about using C/C++ pragma, see z/OS XL C/C++ User’s Guide.

**PL/I**

Runtime options can be specified in a PL/I source application with the following declaration:

```
DCL PLIXOPT CHAR(length) VAR INIT('string')
STATIC EXTERNAL;
```

where string is a list of options separated by commas or blanks, and length is a constant equal to or greater than the length of string. Runtime options in PLIXOPT are parsed by the compiler. For Enterprise PL/I for z/OS and PL/I for MVS & VM, the compilers produce the CEEUOPT CSECT for the PLIXOPT string.

If more than one external procedure in a job declares PLIXOPT as STATIC EXTERNAL, only the first link-edited string is available at run time.

If the PLIXOPT string is specified in an OS PL/I main procedure, the options in the string are processed as if specified in the CEEUOPT CSECT. However, mixing a user-provided CEEUOPT with PLIXOPT is not recommended.

Options specified in the PARM parameter override those specified in the PLIXOPT string.

**Order of precedence**

It is possible for all the methods listed in “Methods available for specifying runtime options” on page 101 to be used for a given application. The order of precedence (from highest to lowest) between option specification methods is:

1. Storage-related options which are set in the storage tuning user exit.
2. Options specified by the assembler user exit (CEEBXITA).
3. Options specified on invocation of the application:
   - Batch: PARM=prog args/rto (COBOL) or PARM=rto/prog args (non-COBOL).
   - TSO: progrname prog args/rto (COBOL) or progrname rto/prog args (non-COBOL).
   - z/OS UNIX: Set _CEE_RUNOPTS environment variable. See Passing environment variables to BPXBATCH (z/OS UNIX System Services User’s Guide) for more information.
   - CICS: Runtime options cannot be specified at transaction invocation.
4. Options specified at invocation time through a DD card (DD:CEEOPTS). The CEEOPTS DD is ignored under CICS, SPC, and for programs invoked using one of the exec() family of functions.
5. Options specified in a CEEUOPT CSECT. There are a few methods available to provide a CEEUOPT CSECT:
   - Assemble a CEEUOPT. For more information, see “Creating application-specific runtime option defaults with CEEXOPT” on page 105.
   - Specify the PL/I PLIXOPT declaration within a source program. The compiler generates the CEEUOPT CSECT from the given options.
   - Specify the C/C++ #pragma runopts() directive within a source program. The compiler generates the CEEUOPT CSECT from the given options.

If you select PLIXOPT or #pragma runopts(), specify it in one and only one compile unit in the application; for example, in the main routine. If multiple CEEUOPTs are present, binder input ordering determines which CEEUOPT is used in an executable program. Only the first CEEUOPT CSECT linked in an executable program is applied. The binder treats any subsequent CEEUOPTs seen in the input as duplicates and they will be ignored.
6. Region-level default options defined within CEEROPT.
7. System-level default options changed after IPL with a SETCEE command.
8. System-level default options changed after IPL with a SET CEE command.
9. System-level default options set in a CEEPROMxx parmlib member and identified during IPL by a CEE=xx statement. This statement can be specified either in the IEASYSyy data set or in the IPL parameters.
10. IBM-supplied defaults.

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

Order of precedence examples

**Example 1 (non-CICS):**

<table>
<thead>
<tr>
<th>IBM-supplied default</th>
<th>CEERPROMxx used at IPL</th>
<th>CEEUOPT</th>
<th>Used at runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE=((NONE, NONE, NONE, 0K), OVR)</td>
<td>STORAGE=((00, NONE, NONE, 0K), OVR)</td>
<td>STORAGE=(00, NONE, 00, 0K)</td>
<td></td>
</tr>
</tbody>
</table>

**Example 2 (CICS):**

<table>
<thead>
<tr>
<th>IBM-supplied default</th>
<th>CEEROPT</th>
<th>CEEUOPT</th>
<th>Used at runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE=((NONE, NONE, NONE, 0K), OVR)</td>
<td>STORAGE=((C1, NONE, NONE, 0K), OVR)</td>
<td>STORAGE=(C2)</td>
<td></td>
</tr>
</tbody>
</table>

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 STACK(,,ANYWHERE,FREE). However, trailing commas are not required; STACK(4K,4K,ANYWHERE) is valid. If you do not specify any suboptions, either of the following is valid: STACK or STACK().

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 slash (/). For more information about program arguments, see “Argument lists and parameter lists” on page 113.

Runtime options usually precede program arguments whenever they are specified in JCL or on application invocation. The possible combinations are described in Table 16 on page 104. You can override this format to ensure compatibility with COBOL applications. See “COBOL compatibility considerations” on page 109 for more information.

<table>
<thead>
<tr>
<th>Table 16: Formats for specifying runtime options and program arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>When the following situations are present:</strong></td>
</tr>
<tr>
<td>Only runtime options are present</td>
</tr>
</tbody>
</table>
Table 16: Formats for specifying runtime options and program arguments (continued)

<table>
<thead>
<tr>
<th>When the following situations are present:</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only program arguments are present</td>
<td>One of the following:</td>
</tr>
<tr>
<td>1. If a slash is present in the arguments, a preceding slash is mandatory.</td>
<td>1. /program arguments</td>
</tr>
<tr>
<td>2. If a slash is not present in the arguments, a preceding slash is optional.</td>
<td>2. program arguments or /program arguments</td>
</tr>
<tr>
<td>Both runtime options and program arguments are present</td>
<td>Runtime options/program arguments</td>
</tr>
</tbody>
</table>

Use the callable service CEE3PRM and CEE3PR2 to retrieve program arguments (for more information, see z/OS Language Environment Programming Reference).

**Restriction:** Program arguments cannot be passed either by a CEEPRMxx parmlib member or by a CEEOPTS DD statement.

In the following example, an object module called MYPROG is created and run using the cataloged procedure CEEWLG. The code in the example overrides the Language Environment defaults for the RPTOPTS and MSGFILE runtime options.

```
//CEEWLG  JOB
//*
//LINKGO EXEC CEEWLG,
//    PARM.GO='RPTOPTS(ON),MSGFILE(OPTRPRT)/'
//*
//LKED.SYSIN   DD DSN='userid.MYLIB.OBJLIB(MYPROG)',...DISP=SHR
//GO.OPTRPRT   DD SYSOUT=A
//*
```

**Creating application-specific runtime option defaults with CEEXOPT**

You can specify a set of application-specific runtime option defaults with the CEEUOPT assembler language source program. When the CEEUOPT source program is assembled, the CEEXOPT macro creates an object module, called CEEUOPT, that can be linked with a program to establish application default options.

The CEE.SCEESAMP data set contains the IBM-supplied sample for the CEEUOPT source program, as shown in Figure 46 on page 106. In the CEEUOPT sample, all runtime options are coded with the IBM-supplied default suboption values. See z/OS Language Environment Programming Reference to select the values appropriate for your application.

The options and suboptions specified in CEEUOPT override the defaults, unless the system-level or region-level defaults were set as nonoverridable (NONOVR). Options specified in CEEUOPT cannot be designated as overridable or nonoverridable.

The CEE.SCEESAMP data set also contains CEEWUOPT, which is the sample job used to assemble the CEEUOPT source program to create the CEEUOPT object module in a user-specified library. CEEWUOPT does not use SMP/E to create the CEEUOPT object module, so it can be run several times to create several different CEEUOPT modules, each in its own user-specified library.
### CEEXOPT invocation for CEEUOPT

To invoke CEEXOPT and create the CEEUOPT object module, follow these steps:

#### Using runtime options

```
CEEUOPT CSECT CEEUOPT AMODE ANY  
CEEUOPT RMODE ANY  
CEEUOPT ABPERC=(NONE), X  
ABTERM=(ABEND), X  
AIXBLD=(OFF), X  
ALL31=(ON), X  
ANYHEAP=(16K,8K,ANYWHERE,FREE), X  
BELOWHEAP=(8K,4K,FREE), X  
CBOPTS=(ON), X  
CBLPSPHP=(ON), X  
CBLQDA=(OFF), X  
CEEDUMP=(60, SYSOUT=*,FREE=END, SPIN=UNALLOC), X  
CHECK=(ON), X  
COUNTRY=(US), X  
DEBUG=(OFF), X  
DEPTHCONDLMT=(10), X  
DYNDUMP=(*USERID,NODYNAMIC,TDUMP), X  
ENVAR=(''), X  
ERRCOUNT=(0), X  
ERRUNIT=(6), X  
FILEHIST=(ON), X  
FILETAG=(NOAUTOCVT,NOAUTOTAG), X  
HEAP=(32K,32K,ANYWHERE,KEEP,8K,4K), X  
HEAPCHK=(OFF,1,0,0,0,1024,0,1024,0), X  
HEAPPools=(OFF,8,10,32,128,10,256,10,1024,10,2048,10,0,10,0,10,0,10,0,10,0,10), X  
HEAPZones=(0,ABEND,0,ABEND), X  
INFOMSGFILTER=(OFF,,,,), X  
INQPCOPN=(ON), X  
INTERRUPT=(OFF), X  
LIBSTACK=(4K,4K,FREE), X  
MSGFILE=(SYSOUT,FBA,121,0,NOENQ), X  
MSGQ=(15) X  
NATLANG=(ENU), X  
NOAUTOTASK=, X  
NOTEST=(ALL,*,PROMPT,INSPPREF), X  
NOUNHDLR=(''), X  
OCSTATUS=(ON), X  
PAGEFRAMESIZE=(4K,4K,4K), X  
PC=(OFF), X  
PLITASKCOUNT=(20), X  
POSIX=(OFF), X  
PROFILE=(OFF,''), X  
PRTUNIT=(6), X  
PUNUNIT=(7), X  
RDRUNIT=(5), X  
RECNRAD=(OFF), X  
RPTOPTS=(OFF), X  
RPTSTG=(OFF), X  
RTERES=(OFF), X  
SIMVRD=(OFF), X  
STACK=(128K,128K,ANYWHERE,KEEP,512K,128K), X  
STORAGE=(NONE, NONE, NONE, 0K), X  
TERMTHDACT=(TRACE,96), X  
THREADHEAP=(OFF,4K,4K,ANYWHERE,KEEP), X  
THREADSTACK=(OFF,4K,4K,ANYWHERE,KEEP,128K,128K), X  
TRACE=(OFF,4K,DUMP,LE=0), X  
TRAP=(ON,SPIE), X  
UPSI=(00000000), X  
VCRTSAVE=(OFF), X  
XPLINK=(OFF), X  
XUFLOW=(AUTO)
```

**Figure 46: Sample Invocation of CEEXOPT within CEEUOPT source program**
1. Copy member CEEUOPT from CEE.SCEESAMP into CEEWUOPT in place of the comment lines following the SYSIN DD statement.

2. Change the parameters on the CEEXOPT macro statement in CEEUOPT to reflect the values you chose for this application-specific runtime options module.

3. Code just the options you want to change. Options omitted from CEEUOPT remains the same as the defaults.

4. Change DSNAME=YOURLIB in the SYSLMOD DD statement to the name of the partitioned data set into which you want your CEEUOPT module to be link-edited.

   **Note:** If you have a CEEUOPT module in your current data set, it is replaced by the new version.

5. Check the SYSLIB DD statement to ensure that the data set names are correct.

   CEEWUOPT should run with a condition code of 0.

**CEEXOPT coding guidelines for CEEUOPT**

Be aware of the following coding guidelines for the CEEXOPT macro:

- A continuation character (X in the source) must be present in column 72 on each line of the CEEXOPT invocation except the last line.

- Options and suboptions must be specified in uppercase. Only suboptions that are strings can be specified in mixed case or lowercase. For example, both MSGFILE=(SYSOUT) and MSGFILE=(sysout) are acceptable. ALL31=(off) is not acceptable.

- 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 was exceeded, and in setting the effective length of the string.

- 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 are properly parsed if this mistake is made.

- 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.

- You can completely omit the specification of any runtime option. Options not specified retain the current default values. There are two other methods available for omitting an option, as follows:

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

    | HEAP=, |
    | X     |

  - Specify the option with empty parentheses and a comma following the equal sign, for example:

    | HEAP=(), |
    | X       |

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

- You can completely omit 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. You can use commas to indicate the omission of one or more suboptions for options having more than one suboption. For
example, if you want 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 |

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 COUNTRY option can be specified in either of the following ways:

| COUNTRY=(US), | X |
| COUNTRY=US, | X |

**Performance considerations**

For optimal performance when using CEEUOPT, code only those options that you want to change. This action enhances performance by minimizing the number of options lines that Language Environment must scan. Options and suboptions that are to remain the same as the defaults do not need to be repeated. For example, if the only change you want to make is to define STACK with an initial value of 64K and an increment of 64K, include only that runtime option, as shown in the following example:

```plaintext
CEEUOPT CSECT
CEEUOPT AMODE ANY
CEEUOPT RMODE ANY
CEEXOPT STACK=(64K,64K)
END
```

**Using the CEEOPTS DD statement**

Language Environment allows you to provide additional invocation-level runtime options using the CEEOPTS DD statement. The CEEOPTS DD can refer to an in-stream data set, regular sequential data set, or a member of a regular or extended partitioned data set. If specified, the data set must be available during initialization of the enclave so the options can be merged. To specify the CEEOPTS DD statement, use the following syntax, as appropriate.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Syntax to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>For in-stream JCL</td>
<td>//CEEOPTS DD * ALL31(OFF),STACK(,,BELOW)</td>
</tr>
<tr>
<td>For a sequential data set</td>
<td>//CEEOPTS DD DSN=MY.CEEOPTS.DATASET,DISP=SHR</td>
</tr>
<tr>
<td>For a partitioned data set</td>
<td>//CEEOPTS DD DSN=MY.CEEOPTS.DATASET(MYOPTS), DISP=SHR</td>
</tr>
<tr>
<td>To ignore the DD statement</td>
<td>//CEEOPTS DD DUMMY</td>
</tr>
</tbody>
</table>

Before using the CEEOPTS DD statement, review the following restrictions:

1. The CEEOPTS DD supports only DASD data sets that can be read with QSAM. An informational message is issued when an unsupported data set type is used.
2. Only the first 3K (excluding comment lines) of the CEEOPTS file are read. All other information is ignored.
3. The file must be in fixed-block or fixed format. Variable block format is not supported.
4. The CEEOPTS DD is ignored under CICS, SPC, and for an exec()ed program.

5. Language Environment cannot use a CEEOPTS DDNAME dynamically allocated with the XTIOT, UCB nocapture, or DSAB-above-the-line options specified in the SVC99 parameters (S99TIOEX, S99ACUCB, S99DSABA flags).

As Figure 47 on page 109 shows, the syntax of the runtime options within the CEEOPTS DD statement is similar to the syntax used on the JCL PARM= parameter or under TSO.

- Each record in the CEEOPTS DD is concatenated to form one options string. No implicit space is added between lines.
- Options must be separated by commas or blanks and can span multiple lines.
- A trailing slash (/) is not valid to denote the end of the options string.
- Input lines that begin with an asterisk (*) in column 1 are treated as comments and ignored.
- The last 8 columns of each record are treated as sequential information and ignored.

```
000001 * This line is a comment
000002 ALL31(OFF),STACK(,,BELOW)
000003 TRAP(ON,
000004   NOSPIE
000005 * This line is a comment within an option
000006 ),TERMTHDACT(
000007 UAIMM,
000008 CICSDDS,96)
```

Figure 47: Example syntax for the CEEOPTS DD statement

The Language Environment runtime options report identifies the options merged from the CEEOPTS DD source by using DD:CEEOPTS as an indicator. For an example of the options report, see z/OS Language Environment Debugging Guide.

### Runtime compatibility considerations

This topic discusses runtime compatibility considerations for C and C++, COBOL, Fortran, PL/I, and IMS.

#### C and C++ compatibility considerations

C and C++ provide 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 `#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.

See z/OS Language Environment Programming Reference for a description of the EXECOPS runtime option.

#### COBOL compatibility considerations

With OS/VS COBOL and VS COBOL II, you must use the following format when specifying the runtime options list:

```
program arguments / runtime options
```

This format is the opposite of the Language Environment-defined format. To ensure compatibility with COBOL, Language Environment provides the runtime option CBLOPTS. With it, you can choose if runtime options or program arguments are expected first in the parameter list. CBLOPTS can only be specified at
Using runtime options

the system level, region level, or in a CEEUOPT CSECT. You can specify a slash (/) as part of the program arguments with CBLOPTS(ON) or CBLOPTS(OFF).

CBLOPTS(ON) allows the existing COBOL format of the invocation character string to continue working (program arguments followed by runtime options). When CBLOPTS(ON) is specified, the last slash in a string delineates the program arguments from the runtime options. Anything before the last slash is interpreted as a program argument.

If there are only invalid runtime options, then the entire string is interpreted as a program argument. For example, if you pass the string 11/16/1967, 1967 is interpreted as an invalid runtime option. Since there are no other runtime options, the entire string is interpreted as a program argument.

Conversely, when CBLOPTS(OFF) is specified, the first slash delineates the runtime options from the program arguments. Anything after the first slash is interpreted as a program argument. CBLOPTS is honored only when a COBOL program is the main routine in the application. For example, if the main routine is C, Language Environment does not honor CBLOPTS. See z/OS Language Environment Programming Reference for more information.

For non-CICS, ensure that COBOL transactions are not link-edited with IGZETUN, which is not supported and which causes an informational message to be logged.

For CICS, ensure that COBOL transactions are not link-edited with IGZEOPT and IGZETUN, which are not supported and which cause an informational message to be logged.

Logging this message for each application inhibits performance.

Fortran compatibility considerations

Under VS FORTRAN Version 2, a slash (/) is not required after runtime options if only runtime options are passed. With Language Environment, however, a slash following runtime options is mandatory. Therefore, you must check your invocation string to ensure the presence of a slash after the runtime options.

There are some differences between Fortran and Language Environment runtime options. While most of these differences are automatically mapped, some options need to be coded in a different format under Language Environment. In addition, there are other Fortran runtime options that are not available under Language Environment. See z/OS Language Environment Programming Reference for the mapping of Fortran to Language Environment runtime options.

If the runtime options string includes an unrecognized option or suboption, Language Environment prints an informational message to help you identify the source of the error.

You can use the Fortran ARGSTR subroutine to retrieve any user-supplied program arguments from the command line. ARGSTR can be used from your Fortran program to identify the program arguments that were given when the enclave was invoked. For information about using ARGSTR in a Fortran program, see VS FORTRAN Version 2 Language and Library Reference.

PL/I compatibility considerations

Under OS PL/I, a slash (/) is not required after runtime options if the runtime options are the only ones passed. With Language Environment, however, a slash is mandatory. Therefore, you must check your invocation string to ensure the presence of a slash after the runtime options.

If a PL/I main program is compiled with the NOEXECOPS option, runtime options cannot be specified in the MVS PARM statement. If runtime options are specified, they are passed as program arguments. The effect of the NOEXECOPS option is described in Appendix D, “Operating system and subsystem parameter list formats,” on page 503.

IMS compatibility considerations

You cannot pass runtime options as CEETDLI arguments, nor can you alter the settings of runtime options when invoking IMS facilities. For more information about using the CEETDLI interface, see “Using the interface between Language Environment and IMS” on page 367.
Running an application is generally the same under Language Environment as in earlier versions of a language's run time. However, to take advantage of some of the features that a common execution environment offers, you must consider a number of different things when preparing an application to run in Language Environment.

When running applications in Language Environment, you must consider the target operating system. Under batch, TSO/E, CICS, and IMS, the way that parameters are passed differs. To ensure consistency, Language Environment standardizes the parameters as much as possible. It is therefore important for you to know what Language Environment does to the format to ensure this consistency. Information about parameter list formats is in Chapter 10, “Using Language Environment parameter list formats,” on page 113 and Appendix D, “Operating system and subsystem parameter list formats,” on page 503.

In addition to describing parameter list formats, this section describes how to manage return codes and offers suggestions on how to make your Language Environment-conforming applications reentrant.
Chapter 10. Using Language Environment parameter list formats

This topic 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. Each Language Environment-conforming HLL might have its own method for transferring control and passing arguments between internal routines.

When writing a Language Environment-conforming 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 \texttt{argv} and \texttt{argc}. If you set up your C, C++, COBOL, or PL/I 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.

Fortran does not support passing parameters to a main routine.

On operating subsystems such as CICS and IMS, however, the parameter format that is passed might be different from what your main routine expects. In these cases, you must explicitly code your main routine to accept the format of the parameters as they are passed by CICS and IMS.

“Preparing your main routine to receive parameters” on page 116 contains examples of how to code your main routine to receive parameters under any supported operating system or subsystem.

Additionally, some HLLs, such as C, C++, and PL/I, provide options that enable you to specify the format of the parameter list you expect to be passed to your main routine. For example, C programmers can specify the PLIST runtime option, which determines the parameter list format. If your HLL provides such an option, refer to one of the following for information about which settings you should select to run an application:

- **TSO/E**
  - Table 19 on page 116
- **IMS**
  - Table 20 on page 118
- **CICS**
  - Table 21 on page 118
- **MVS**
  - Table 22 on page 119

When running most main routines, you do not need to explicitly access the parameter list. Language Environment provides the CEE3PRM and CEE3PR2 callable service to query and return to your calling routine the parameter string passed to your main routine when it was invoked. The returned parameter string contains only program arguments. If no program arguments were specified, a blank string is returned. See \textit{z/OS Language Environment Programming Reference} for more information about CEE3PRM and CEE3PR2.

In addition, some HLLs, such as C and C++, provide ways of identifying passed parameters to your main routine using constructs within the HLL itself. For more information, see “C and C++ parameter passing considerations” on page 503.

**Argument lists and parameter lists**

The terminology used to describe passing parameters to and from routines currently differs among Language Environment-conforming HLLs. Figure 48 on page 114 summarizes the terminology used with
Language Environment. In Figure 48 on page 114, 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 48: 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 that is passed by the calling routine.

**By reference**
- Changes that are made by the called routine to the argument value can alter the original argument value that is 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 17 on page 114 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>
<thead>
<tr>
<th>Term</th>
<th>By value</th>
<th>By reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>The value of the object is passed</td>
<td>Not allowed under Language Environment</td>
</tr>
<tr>
<td>Indirect</td>
<td>A pointer points to the value of an object</td>
<td>A pointer points to the object</td>
</tr>
</tbody>
</table>

Figure 49 on page 115 illustrates these argument passing styles. In Figure 49 on page 115, 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.
Figure 49: Argument passing styles in Language Environment

HLL semantics usually determine when data is passed by value or by reference. Language Environment supports argument passing styles as shown in Table 18 on page 115.

Table 18: Default passing style per HLL.

<table>
<thead>
<tr>
<th>Language</th>
<th>Default argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (including XPLINK). See note 4.</td>
<td>By Value (Direct)</td>
</tr>
<tr>
<td>COBOL</td>
<td>By Reference (Indirect) (COBOL BY REFERENCE). See note 2.</td>
</tr>
<tr>
<td>Fortran</td>
<td>By Reference (Indirect)</td>
</tr>
</tbody>
</table>

Notes:

1. C++ also supports by reference (indirect), if a prototype specifies it with an ampersand (&).
2. Other parameter passing styles that are supported are:
   - By value (Indirect) (COBOL BY CONTENT) by (COBOL BY CONTENT) by Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II
   - By value (Direct) (COBOL BY VALUE) by Enterprise COBOL for z/OS, COBOL for OS/390 & VM, and COBOL for MVS & VM
3. However, when SYSTEM(CICS) or SYSTEM(IMS) is specified, Enterprise PL/I for z/OS and PL/I for MVS & VM main procedures assume by value (direct) for parameters (OS PL/I main procedures continue to assume by reference (indirect)). (See “PL/I argument passing considerations” on page 119 for a discussion of OPTIONS(BYVALUE).)

PL/I and Fortran also support by value (indirect) (also known as by content), which you can obtain by passing an argument in parentheses, for example, A in CALL X(A,B).
4. XPLINK-compiled functions pass arguments by value by default. However, it will pass as many arguments in registers as possible in order to reduce the call linkage overhead. See z/OS Language Environment Vendor Interfaces for additional information.

Preparing your main routine to receive parameters

When coding a main routine to receive a parameter list from the operating system, consider the following items:

- The HLL in which your main routine is written
  HLL semantics determine how you code your main routine in order to receive a parameter list.
- The method of main routine invocation
  You should consider the environment (MVS, TSO, IMS, CICS) in which your main routine is invoked, as well as the commands used to invoke it.
- The compiler or runtime options that you must specify
  The settings of the C PLIST runtime option, the C++ PLIST compiler option, or the PL/I SYSTEM compiler option that you must specify are based on:
  - The operating system or subsystem where you invoke your main routine
  - The commands you use to invoke your main routine

The following tables summarize options to consider when preparing a main routine to receive parameters in each system or subsystem; the tables also provide sample coding for each HLL:

**TSO/E**
Table 19 on page 116

**IMS**
Table 20 on page 118

**CICS**
Table 21 on page 118

**MVS**
Table 22 on page 119

---

**Table 19: Coding a main routine to receive an inbound parameter list in TSO/E.**

<table>
<thead>
<tr>
<th>Language</th>
<th>Recommended options setting</th>
<th>Sample main routine code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C or C++ (See note 1.)</td>
<td>In C, specify PLIST(HOST) runtime option; if not specified, PLIST(HOST) is on by default. Under C++, this is the default behavior; do not specify a PLIST compiler option setting.</td>
<td>main(int argc, char * argv[])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Language</td>
<td>Recommended options setting</td>
<td>Sample main routine code</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| C or C++ (See note 2.) | In C, PLIST(HOST) is the default; PLIST(TSO) is supported for compatibility and acts the same as PLIST(HOST). argc and argv are set from the command buffer. In C++, this is the behavior by default. To see the TSO CPPL, specify PLIST(OS) and access the CPPL through __osplist. In C++, you must specify the PLIST(OS) compiler option. | For PLIST(HOST) behavior, see above. The following code is a sample that accesses the TSO CPPL:  
```c
#include <stdlib.h>  
typedef struct CPPL_STRUCT {  
  void * CPPLCBUF;  
  void * CPPLUPT;  
  void * CPPLPSCB;  
  void * CPPLECT;  
} CPPL;  
main()  
{  
  CPPL *cppl_ptr;  
  cppl_ptr = __osplist;  
  ;  
}  
``` |
| COBOL (See note 1.) | No specific options required.                                                                 | IDENTIFICATION DIVISION.  
```cobol
IDENTIFICATION DIVISION.  
::  
DATA DIVISION.  
::  
LINKAGE SECTION.  
 01 PARMDATA.  
 02 STRINGLEN PIC 9(4) USAGE IS BINARY.  
 02 STR.  
 03 PARM-BYTE PIC X OCCURS 0 TO 100 DEPENDING ON STRINGLEN.  
::  
PROCEDURE DIVISION USING PARMDATA.  
::  
``` |
| COBOL (See note 2.) | No specific options required.                                                                 | Same as above. |
| PL/I (See note 1.) | Specify SYSTEM(MVS) compiler option.                                                           | *PROCESS SYSTEM(xxx);  
```pli
MYMAIN: PROC (A) OPTIONS (MAIN);  
  DCL A CHAR(100) VARYING;  
  ;  
``` |
| PL/I (See note 2.) | Specify SYSTEM(TSO) compiler option.                                                            | *PROCESS SYSTEM(TSO);  
```pli
MYMAIN: PROC (CPPLPTR) OPTIONS (MAIN);  
  /*Pointer to CPPL*/  
  DCL CPPLPTR POINTER;  
  DCL 1 CPPL based (CPPLPTR),  
  /*Command buffer*/  
  2 CPPLCBUF POINTER,  
  /*User profile table*/  
  2 CPPLUPT POINTER,  
  /*Protected step ctl blk*/  
  2 CPPLPSCB POINTER,  
  /*Environment ctl blk*/  
  2 CPPLECT POINTER;  
  ;  
``` |

**Method of invocation:**

1. Use the LOADGO command or the CALL command.
2. Use the TSO Command Processor.

Table 20: Coding a main routine to receive an inbound parameter list in IMS

<table>
<thead>
<tr>
<th>Language</th>
<th>Recommended options setting</th>
<th>Sample main routine code</th>
</tr>
</thead>
</table>
| C        | Specify PLIST(OS) and ENV(IMS) runtime option. | ```
#pragma runopts(env(ims),plist(os))
#include <ims.h>
typedef struct {PCB_STRUCT(10)} PCB_10_TYPE;
main()
{
   PCB_STRUCT_8_TYPE *alt_pcb;
   PCB_10_TYPE *db_pcb;
   IO_PCB_TYPE *io_pcb;
   ...
}
``` |
| C++      | Specify PLIST(OS) and TARGET(IMS) compiler option. | ```
#include <ims.h>
typedef struct {PCB_STRUCT(10)} PCB_10_TYPE;
main()
{
   PCB_STRUCT_8_TYPE *alt_pcb;
   PCB_10_TYPE *db_pcb;
   IO_PCB_TYPE *io_pcb;
   ...
}
``` |
| COBOL    | No specific options required. | ```
IDENTIFICATION DIVISION.
PROGRAM-ID. DLITCBL.
DATA DIVISION.
...
LINKAGE SECTION.
  01 PCB1.
      02 ...
  01 PCB2.
      02 ...
...
PROCEDURE DIVISION USING PCB1, PCB2.
``` |
| PL/I      | Specify SYSTEM(IMS) compiler option. | ```
*PROCESS SYSTEM(IMS);
MYMAIN: PROC (X,Y,Z) OPTIONS(MAIN);
   DCL (X,Y,Z) POINTER;
   DCL 1 PCB based (X),
   ...
``` |

Table 21: Coding a main routine to receive an inbound parameter list in CICS

<table>
<thead>
<tr>
<th>Language</th>
<th>Recommended options setting</th>
<th>Sample main routine code</th>
</tr>
</thead>
</table>
| C or C++ | Do not specify any PLIST option. argc = 1 and argv[0] = transaction id. | ```
main(int argc,char *argv[])
{
...
``` |
### Table 21: Coding a main routine to receive an inbound parameter list in CICS (continued)

<table>
<thead>
<tr>
<th>Language</th>
<th>Recommended options setting</th>
<th>Sample main routine code</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOL</td>
<td>No specific options required.</td>
<td>IDENTIFICATION DIVISION.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA DIVISION.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LINKAGE SECTION.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 DFHEIBLK.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 DFHCOMMAREA.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROCEDURE DIVISION USING DFHEIBLK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DFHCOMMAREA.</td>
</tr>
<tr>
<td>PL/I</td>
<td>Specify SYSTEM(CICS) compiler option.</td>
<td>*PROCESS SYSTEM(CICS);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MYMAIN: PROC (DFHEIPTR,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DFHCOMMAREAPTR_PTR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OPTIONS(MAIN);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/<em>pointer to EIB</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/<em>supplied by CICS translator</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DCL DFHEIPTR POINTER;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/<em>pointer to commarea</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DCL DFHCOMMAREAPTR_PTR POINTER;</td>
</tr>
</tbody>
</table>

### Table 22: Coding a main routine to receive an inbound parameter list in MVS. Method of invocation: Assembler passing an arbitrary parameter list that Language Environment is not to interpret.

<table>
<thead>
<tr>
<th>Language</th>
<th>Recommended options setting</th>
<th>Sample main routine code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C or C++</td>
<td>In C, specify PLIST(OS) runtime option.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In C++, specify PLIST(OS) compiler option.</td>
<td></td>
</tr>
<tr>
<td>COBOL</td>
<td>No specific options required.</td>
<td>IDENTIFICATION DIVISION.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA DIVISION.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LINKAGE SECTION.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 PARM1...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 PARM2...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROCEDURE DIVISION USING PARM1, PARM2.</td>
</tr>
<tr>
<td>PL/I</td>
<td>Specify SYSTEM(MVS) and NOEXECOPS procedure option.</td>
<td>*PROCESS SYSTEM(MVS);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MYMAIN: PROC (PARM1,PARM2,...)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OPTIONS (MAIN NOEXECOPS);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DCL PARM1...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DCL PARM2...</td>
</tr>
</tbody>
</table>

**PL/I argument passing considerations**

The PL/I OPTIONS option of both the PROCEDURE statement and ENTRY declaration permits you to specify the mutually exclusive options BYVALUE and BYADDR.
OPTIONS(BYVALUE)
Specifies that the PL/I procedure expects arguments to be passed to it by value (direct).
OPTIONS(BYVALUE) can be specified for external PROCEDURE statements and ENTRY declarations. It applies to all arguments and argument descriptors.

OPTIONS(BYADDR)
Specifies that the PL/I procedure expects arguments to be passed to it by reference (indirect) or by value (indirect). OPTIONS(BYADDR) can be specified for external PROCEDURE statements and for ENTRY declarations. It applies to all arguments and argument descriptors.

OPTIONS(BYVALUE) cannot be specified for the following constructs:
• ENTRY statements:
  ENTRY(N) OPTIONS(BYVALUE); /* invalid */
• Declaration of a parameter:
  PROC(ARG1);
  DCL ARG1 FIXED BIN(31) BYVALUE; /* invalid */
• Parameter descriptor in an ENTRY declaration:
  DCL T ENTRY(FIXED BIN(31) BYVALUE) EXTERNAL; /* invalid */

All parameters, parameter descriptors, or return values must be specified with either the POINTER or FIXED BIN(31) data type. Return values are passed back in register 15.

OPTIONS(BYADDR) is the default unless the external procedure specifies OPTIONS(MAIN) and is compiled with the SYSTEM(CICS) or SYSTEM(IMS) compiler option. In this case, OPTIONS(BYVALUE) is the default. In general, you should specify OPTIONS(BYVALUE) only for a main procedure with a SYSTEM option of IMS or CICS. If you specify OPTIONS(BYVALUE) for a main procedure with other system options, the parameter list is passed to the main procedure as is.

OPTIONS(BYVALUE) for a main procedure implies OPTIONS(NOEXECOPS).

PL/I does not support calls to routines that modify the body of an indirect argument list built by PL/I compiled code.
Chapter 11. Making your application reentrant

This topic shows you how to make your 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.

The following routines must be reentrant:

- Routines to be loaded into the LPA or ELPA
- Routines to be used with CICS
- Routines to be preloaded with IMS

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 123.

### Constructed reentrancy

A constructed reentrant program is created by using either of the following methods:

- Use the binder to combine all of the object modules produced by the z/OS XL C/C++ compiler when the target data set is a PDSE or HFS.
- Use the prelinker to combine all of the object modules produced by the z/OS XL C/C++ compiler and pass the output to the binder when the target data set is a PDS. For more information about the prelinker see Appendix A, “Prelinking an application,” on page 481.

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 either the binder or the prelinker), which is initialized at run time. The second part can be shared across multiple spaces or sessions only if it is installed in the link pack area (LPA) or extended link pack area (ELPA).

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 compiler option. For C++, compile your source files; 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:
   - If the target data set is a PDSE or HFS, use the binder to combine all of the input into an executable program
   - If the target data set is a PDS, use the prelinker to combine all of the input before passing it as input to the binder. You cannot run an object module through the prelinker more than once. Also, you must link-edit using the same platform you used for the prelink step.
3. To get the greatest benefit from reentrancy, install your executable program in one of the locations listed in “Installing a reentrant load module” on page 123.

Making your COBOL program reentrant

If you intend to have multiple users execute a COBOL program concurrently, make it reentrant by using the RENT compiler option. For information about specifying the RENT compiler option, see the appropriate version of the COBOL programming guide in the COBOL library at Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733).

Making your Fortran program reentrant

If you intend to have multiple users execute a Fortran program concurrently, make it reentrant by using the RENT compiler option. The object module produced by the compiler must then be separated into its nonshareable and shareable parts using the reentrancy separation tool.

The Fortran reentrancy separation tool is delivered under Language Environment, and with the exception of its name and the names of the cataloged procedures used to invoke it, its use and operation are the same as with the reentrancy separation tool provided by VS FORTRAN Version 2.

<table>
<thead>
<tr>
<th>Fortran Member name</th>
<th>Language Environment member name</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFBVSFST</td>
<td>AFHXFSTA</td>
<td>Fortran reentrancy separation tool</td>
</tr>
<tr>
<td>None</td>
<td>AFHWRL</td>
<td>Cataloged procedure to separate the nonshareable and shareable parts of an object module, and link-edit</td>
</tr>
<tr>
<td>VFT2RLG</td>
<td>AFHWRLG</td>
<td>Cataloged procedure to separate the nonshareable and shareable parts of an object module, link-edit, and execute</td>
</tr>
</tbody>
</table>

The Fortran reentrancy separation tool is a member of the CEE.SCEERUN data set. The Fortran reentrancy separation tool cataloged procedures are members of the CEE.SCEEPROC data set.
It is important to note that Fortran products from VS FORTRAN Version 1 Release 4 on produce reentrant object code; however, mixing Fortran object code with another HLL's object code can cause the other HLL's load module to become nonreentrant. This is due to the mechanism that Fortran uses to produce reentrant code.

For more information about creating reentrant Fortran programs, see VS FORTRAN Version 2 Programming Guide for CMS and MVS.

Making your PL/I program reentrant

If you intend to have multiple users execute a PL/I program at the same time, make it reentrant by specifying the REENTRANT procedure option when you compile. For information about specifying the REENTRANT procedure option, see PL/I for MVS & VM Language Reference.

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 (LPA) or the extended link pack area (ELPA) of the system.

Installing a module in the LPA, ELPA or saved segment requires an initial program load (IPL) of your operating system. You can use the SET PROG=xx console command to add or remove modules from dynamic LPA.
This section provides more information about Language Environment and the services it provides.
Chapter 12. Initialization and termination under Language Environment

This topic describes initialization and termination under Language Environment. It describes how you can customize your applications during initialization and termination by using Language Environment runtime options, callable services, and user exits. It includes instructions on how to use return and abend codes to respond to initialization and termination actions, as well as to conditions that remain unhandled.

The basics of initialization and termination

Initialization and termination establish the state of various parts of the Language Environment program management model that supports multilanguage applications. The program management 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 at least one main routine.

Thread
The basic unit of execution.

The z/OS UNIX System Services (z/OS UNIX) program management model differs somewhat from the Language Environment program management model. Refer to “Mapping the POSIX program management model to the Language Environment program management model” on page 143 for more information. For more detailed definitions of program management and other Language Environment terms, see Chapter 13, “Program management model,” on page 139.

When you run a routine, Language Environment initializes the runtime environment by creating a process, an enclave, and an initial thread. You can modify initialization by running a user exit, written either in assembler or in an HLL.

During termination, threads (either single or multiple, depending on whether your application is POSIX-conforming), enclaves, and processes are terminated. Through the runtime options of Language Environment and callable services for termination, you can control how a thread, enclave, or process terminates. For example, you can control whether an abend or a return code is generated from an application that terminates with an unhandled condition of severity 2 or greater. See “Termination behavior for unhandled conditions” on page 135.

Related runtime options:

ABTERMENC
Specifies whether an enclave terminates with an abend or with a return code and a reason code when there is an unhandled condition of severity 2 or greater

TERMTHDACT
Specifies the level of information that you want to receive after an unhandled condition of severity 2 or greater causes a thread to terminate

Related callable services:

CEE3ABD
Terminates an enclave with or without clean-up and the value of clean-up specifies which dumps to take during termination.

CEE3AB2
Terminates an enclave with or without clean-up, whose value specifies which dumps to take during termination, and a user specified reason code.
CEE3GRC
Returns the user enclave return code to your routine. Along with CEE3SRC, it allows you to use return code-based programming techniques.

CEE3PRM
Returns to your routine the parameter string specified when your application was invoked. Use CEE3PR2 for parameter strings greater than 80 characters.

CEE3PR2
Returns to the calling routine the argument string and its associated length, specified at program invocation.

CEE3SRC
Sets the user enclave return code, which is used to calculate the final enclave return code at termination

Related user exits:

CEEBXITA
An assembler user exit for enclave initialization, and enclave and process termination

CEEBINT
An HLL user exit (written in C, C++ (with C linkage), Enterprise PL/I for z/OS or PL/I for MVS & VM, or Language Environment-conforming assembler) called at enclave initialization

See Chapter 28, “Using runtime user exits,” on page 373 for more information about user exits.

Preinitialization interface:

CEEPIPI
CEEPIPI performs various initialization functions

See Chapter 30, “Using preinitialization services,” on page 429 for more information about the preinitialization interface.

See z/OS Language Environment Programming Reference for syntax information about runtime options and callable services.

Language Environment initialization

During initialization, a process, an enclave, and then an initial thread are created. You can affect initialization at the enclave level, by using either the assembler or HLL user exits.

Process initialization sets up the framework to manage enclaves and initializes resources that can be shared among enclaves. 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.

What happens during initialization

When you run an application under Language Environment, the following sequence of events occurs:

1. Language Environment runs the assembler user exit CEEBXITA.

   CEEBXITA runs before initialization of the enclave.

   You cannot code the CEEBXITA assembler user exit as an XPLINK application. However, since CEEBXITA is called directly by Language Environment and not by the application, a non-XPLINK CEEBXITA can be statically bound in the same program object with an XPLINK application.

   You can modify the environment in which your application runs by:
   • Specifying certain runtime options.
   • Allocating data sets and files.
   • Listing abend codes to be passed to the operating system.
• Checking the values of program arguments.

IBM provides a default version of CEEBXITA and several samples you can use to customize your application to perform tasks such as enforcing a set of runtime options for a particular environment. Because CEEBXITA runs before any HLLs have been established, it is written in assembler language so that it can establish parameters such as stack size and trap settings for the HLLs.

CEEBXITA can function as application-specific or installation-wide. If you customize CEEBXITA to do application-specific processing (for example, dynamically allocating files needed by your application), you must link the exit with the application load module. (Conversely, installation-wide user exits must be linked with the Language Environment initialization library routines.)

An application-specific user exit has priority over an installation-wide exit, so you can customize a user exit for a particular application without affecting the installation default version.

For more information about the function and location of the CEEBXITA user exit, see Chapter 28, “Using runtime user exits,” on page 373.

2. Language Environment examines the load module and initializes all languages identified in the application.

Under Language Environment, an interlanguage communication (ILC) application works as shown in Figure 50 on page 129. Language Environment will also examine the load module and initialize an XPLINK environment (forcing the XPLINK(ON) runtime option) if the initial program was compiled with the XPLINK option. Because all the language conventions are already established and do not need to be initialized and terminated between calls to other routines, the processing is significantly faster when using Language Environment-conforming HLLs.

Performance consideration: Language Environment initializes all languages included in an application, regardless of whether all of them are used. To optimize performance, include only those languages your application actually uses.

3. Language Environment runs the HLL user exit CEEBINT.

CEEBINT lets you perform tasks such as recording accounting statistics or calling other user exits. You cannot code CEEBINT as an XPLINK application. However, since CEEBINT is called directly by Language Environment and not the application, a non-XPLINK CEEBINT can be statically bound in the same program object with an XPLINK application. You can write a customized version of CEEBINT in any Language Environment-conforming language except COBOL. COBOL applications can, however, use CEEBINT written in another language.

IBM provides an object module default version of CEEBINT that consists of an immediate return to the application. This default version is automatically link-edited with your application unless you provide a customized version of CEEBINT.

For more information about the function and location of the CEEBINT user exit, see Chapter 28, “Using runtime user exits,” on page 373.
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, callable services, and user exits. For example, if an unhandled condition of severity code 2 or greater occurs, you can decide if Language Environment should issue a return code or an abend code to the application. See “Termination behavior for unhandled conditions” on page 135 for more information.

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; that is, an implicit STOP is performed.
- 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
  - COBOL's STOP RUN statement
  - COBOL's GOBACK statement in a main program
  - Fortran's STOP statement
  - Fortran's CALL SYSRCX, CALL EXIT, CALL DUMP, or CALL CDUMP statement
  - PL/I's STOP or 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 CEE3ABD or CEE3AB2).
- An unhandled condition of severity 2 or greater occurs. (See “Termination behavior for unhandled conditions” on page 135 for information.)

What happens during termination

The following sequence of events occurs during termination:

1. `atexit()` functions are invoked, if present. They are not invoked if `_exit` calls for termination or if abnormal termination occurs. The behavior of pthread functions are undefined if the pthread functions are called from an atexit routine.
2. PL/I FINISH ON-units are invoked if established.
3. For normal termination, the enclave return code is set (see “Managing return codes in Language Environment” on page 132). For abnormal termination caused by an unhandled condition of severity 2 or greater, either a return code and reason code or an abend is returned, based on settings specified in CEEBXITA (see “Termination behavior for unhandled conditions” on page 135).
4. CEEBXITA is invoked for enclave termination after all application code has completed, but before any enclave resources are relinquished.

You can modify CEEBXITA to request an abend and a dump. You cannot code the CEEBXITA assembler user exit as an XPLINK application. Because the environment is still active, the dump accurately reflects the state of the environment before an enclave is terminated.
5. The environment is terminated:

- All enclaves are terminated
- All enclave resources are returned to the operating system
- Any files that Language Environment manages are closed
IBM z/OS Debugger is terminated, if active

6. CEEBXITA is invoked for process termination after the environment is terminated. You can modify CEEBXITA to close files, request an abend, or request a dump. A dump requested at this point, however, does not have the degree of detail that one requested during enclave termination has.

CEEBXITA is not invoked for process termination if there is an unhandled condition of severity 2 or greater, or if CEEBXITA requests an abend during enclave termination. For more information about the CEEBXITA assembler user exit, see Chapter 28, “Using runtime user exits,” on page 373.

Depending on the setting of the TERMTHDACT runtime option, you might receive a message, a trace of the active routines, or a dump when a condition of severity 2 or greater occurs. See z/OS Language Environment Programming Reference for more information about TERMTHDACT.

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 131 for information about enclave termination.

POSIX thread termination

A thread terminates due to pthread_exit(), pthread_kill(), or pthread_cancel(), or 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
- Language Environment user-written condition handlers are run, if present
- The thread status is set
- Cleanup handlers and destructor routines are driven
- The stack is collapsed
- HLL members are called for thread termination

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

- “Language Environment and POSIX signal handling interactions” on page 199
- “Mapping the POSIX program management model to the Language Environment program management model” on page 143
- 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:

- Calls HLL-specific termination routines for HLLs that were active during the executing of the program
- Runs Language Environment user-written condition handlers, if present
- Deletes modules loaded by Language Environment
- Frees all storage obtained by Language Environment services
- Calls the CEEBXITA assembler user exit for enclave termination
- Frees Language Environment control blocks for the enclave
- Depending on the setting in the HLL or assembler user exit, 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 occurs when the last enclave in the process terminates. Process termination deletes the structure that kept track of the enclaves within the process, releases the process control block (PCB) and associated resources, and returns control to the creator of the process.

Because Language Environment generally supports a single enclave running within a single process, termination of the enclave means that your application has terminated. For exceptions to the single enclave within a single process and an enclave return and reason code being returned to the invoker, see Chapter 31, “Using nested enclaves,” on page 469.

Language Environment explicitly relinquishes all resources it gets. Routines that get 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 about POSIX default signal action at POSIX process termination when running in a z/OS UNIX environment, see “Language Environment and POSIX signal handling interactions” on page 199.

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 topic 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.)

Before Language Environment, some HLLs (in particular, C) handled conditions that occur in the runtime environment by using a return code-based model. Such a model typically allows return codes to be passed between called subroutines and from the main routine back to the operating system to communicate the status of requested operations. Language Environment, on the other hand, uses a condition-based model to communicate conditions, as described in Chapter 18, “Using condition tokens,” on page 233.

Although Language Environment supports applications that rely on passing return codes from called subroutines and checking these return codes, you are encouraged to use Language Environment condition handling mechanisms, such as user-written condition handlers, instead.

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 return code modifier). The Language Environment enclave return code is calculated by summing the user return code generated by the HLL (see “Setting and altering user return codes” on page 133) and the enclave reason code (see “How the enclave reason code is calculated” on page 135) as follows:

\[
\text{Language Environment enclave return code} = \text{user return code} + \text{enclave reason code}
\]
The Language Environment enclave return code is placed in register 15, and the enclave reason code is placed in register 0.

**C considerations**

The Language Environment enclave return codes are incompatible with the return codes returned under the pre-Language Environment-conforming version of C.

**Fortran considerations**

Unlike the behavior of VS FORTRAN Version 2, where any abnormal termination is indicated with message AFB240 followed by an abend with user completion code 240, Language Environment treats an abend as a condition. The condition that represents an abend is the severity 3 condition with the message number 3250, which contains the system or user completion code and the reason code.

If this condition is not handled and the ABTERMENC(RETCODE) runtime option is in effect, then the enclave terminates with a return code of 3000 under MVS. When the ABTERMENC(ABEND) runtime option is in effect and Language Environment terminates the enclave because of an unhandled condition, an abend occurs.

**PL/I considerations**

The severities of some PL/I conditions have been redefined from what they were in pre-Language Environment versions of PL/I. See z/OS Language Environment Runtime Messages for details.

**Setting and altering user return codes**

User return codes can be set and altered by the CEE3SRC callable service and by language constructs. As described in the following topics, 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. In either case, any user return codes set through CEE3SRC are ignored; likewise, in an ILC application, any user return codes set with PL/I language constructs are also ignored.

If the enclave terminates due to an unhandled condition of severity 2 or greater, the user return code value used is the last one set by either CEE3SRC or, in an ILC application, PL/I language constructs. If neither CEE3SRC nor PL/I language constructs set the user return code, the user return code value is 0. See “Termination behavior for unhandled conditions” on page 135 for information about unhandled conditions. See z/OS XL C/C++ Programming Guide for more information about C or C++ language constructs.

**For COBOL**

When a COBOL program initiates enclave termination, such as with a STOP RUN statement or a GOBACK statement in a main program, the user return code value is taken from the RETURN-CODE special register; any user return codes set through CEE3SRC are ignored. Likewise, in an ILC application, any user return codes set with PL/I language constructs are also ignored. Thus, you can set and alter the user return code and pass it across program boundaries in register 15. See the appropriate version of the programming guide in the COBOL library at Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733) for more information about the RETURN-CODE special register and COBOL language constructs.

If the enclave terminates due to an unhandled condition with severity 2 or greater, the RETURN-CODE special register is not used in the enclave return code calculation. Instead, the user return code value used is the last one set by either CEE3SRC or, in an ILC application, PL/I language constructs. If neither CEE3SRC nor PL/I language constructs have been used to set the user return code, the user return code value is 0. See “Termination behavior for unhandled conditions” on page 135 for information about unhandled conditions.
For Fortran

You can set and alter the user return code using the SYSRCS or SYSRCX Fortran services. You can test the value of this field using SYSRCT. Depending on how the enclave is terminated, the value of the user return code could become the enclave return code.

If the enclave terminates as a result of a STOP statement or a CALL SYSRCX statement that explicitly specifies a value to be used as a return code, then that value becomes the enclave user return code. For example, either of the following Fortran statements terminate the enclave and sets the enclave return code to 101:

- STOP 101
- CALL SYSRCX(101)

If the enclave terminates as a result of a language construct that depends on a previously established enclave user return code, then the previously established enclave user return code becomes the enclave return code. For example, the following sequence of Fortran statements sets the enclave return code to 201:

- CALL SYSRCS(201)
- CALL EXIT

The call to SYSRCS sets the enclave user return code to 201 and the call to EXIT causes the enclave user return code to be used as the enclave return code.

The return code modifier depends on the operating system and the severity of the condition, as shown in Table 24 on page 134.

<table>
<thead>
<tr>
<th>Condition severity</th>
<th>Return code modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>3000</td>
</tr>
<tr>
<td>4</td>
<td>4000</td>
</tr>
</tbody>
</table>

If the enclave terminates due to an unhandled condition and the ABTERMENC(RETCODE) runtime option is in effect, then the enclave return code is the sum of the enclave user return code and the return code modifier. For example, when CALL SYSRCS(201) is executed and termination occurs as a result of an unhandled condition of severity 3, the enclave return code is 3201.

For PL/I

You can set and alter the user return code with the PLIRETC function or the OPTIONS(RETCODE) attribute. The PLIRETV function retrieves the current value of the user return code.

When a PL/I routine initiates enclave termination, such as with a STOP or EXIT statement in a subroutine or with a RETURN or END statement in a main procedure, the user return code is the value set with the PLIRETC function or the OPTIONS(RETCODE) attribute. However, CEE3SRC can alter the user return code set with PLIRETC or the OPTIONS(RETCODE) attribute. If CEE3SRC was the last method used to set the user return code, the last three bytes of the return-code set by CEE3SRC are used as the user return code.

If the enclave terminates due to an unhandled condition with severity 2 or greater, the user return code value set last (with either PL/I constructs or CEE3SRC) is used in the calculation of the enclave return code; if one has not been set, the user return code value is 0. See “Termination behavior for unhandled conditions” on page 135 for information about unhandled conditions.

CEE3SRC and CEE3GRC are not supported in PL/I multitasking applications.

See IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735) for details about the PL/I language constructs.
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 25 on page 135. For more information about Language Environment conditions and severity codes, see Table 34 on page 174.

Table 25 on page 135 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>
<thead>
<tr>
<th>Condition severity</th>
<th>Meaning</th>
<th>Enclave reason code (R0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Normal application termination</td>
<td>0</td>
</tr>
<tr>
<td>Severity 1 condition</td>
<td>Termination_Imminent due to STOP</td>
<td>1000</td>
</tr>
<tr>
<td>Unhandled severity 2 condition</td>
<td>Error — abnormal termination</td>
<td>2000</td>
</tr>
<tr>
<td>Unhandled severity 3 condition</td>
<td>Severe error — abnormal termination</td>
<td>3000</td>
</tr>
<tr>
<td>Unhandled severity 4 condition</td>
<td>Critical error — abnormal termination</td>
<td>4000</td>
</tr>
</tbody>
</table>

Termination behavior for unhandled conditions

When there is an unhandled condition of severity 2 or greater, you can choose whether an enclave terminates with an abend or with a return code and a reason code. Language Environment will assign an abend code and return and reason code, as described in this topic, or you can assign values yourself, as described in “Setting and altering user return codes” on page 133.

See Table 34 on page 174 for a discussion of conditions and how they are handled in Language Environment, and “Language Environment and POSIX signal handling interactions” on page 199 for specific information pertaining to POSIX signal action defaults and unhandled conditions in a z/OS UNIX environment.

Some users, especially those using COBOL or running IMS applications, expect to receive an abend when an error is detected rather than a return code and a reason code. To get this behavior, they can use the ABTERMENC(ABEND) runtime option discussed in “Abend codes generated by ABTERMENC(ABEND) runtime option” on page 136. Other users, however, expect to receive a return code and a reason code when there is an error.

If you are running in a CICS environment, the IBM-supplied default is to terminate the enclave with an abend for unhandled conditions of severity 2 or greater.

If you are running in a non-CICS environment and you expect the enclave to terminate with a return code and a reason code for unhandled conditions of severity 2 or greater, you can use the ABTERMENC(RETCODE) runtime option and the CEEBXITA assembler user exit. The default version of CEEBXITA for non-CICS environments requests that the enclave terminate with a return code and a reason code.

Table 26 on page 136 shows the various types of enclave termination that occur based on the ABTERMENC runtime option settings and the CEEAUE_ABND flag settings of CEEBXITA. See “CEEBXITA assembler user exit interface” on page 378 for an explanation of the CEEAUE_ABND flag.
Table 26: Termination behavior for unhandled conditions of severity 2 or greater

<table>
<thead>
<tr>
<th>ABTERMENC suboption</th>
<th>Value of CEEAUE_ABND flag enclave termination</th>
<th>Enclave termination type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETCODE</td>
<td>0</td>
<td>Return to caller with return code and reason code</td>
</tr>
<tr>
<td>RETCODE</td>
<td>1</td>
<td>Abend using CEEAUE_RETC and CEEAUE_RSNC</td>
</tr>
<tr>
<td>ABEND</td>
<td>0</td>
<td>Abend using the abend codes listed in Table 28 on page 136</td>
</tr>
<tr>
<td>ABEND</td>
<td>1</td>
<td>Abend using CEEAUE_RETC and CEEAUE_RSNC</td>
</tr>
</tbody>
</table>

Determining the abend code

You can choose the abend code you want Language Environment to use, based on whether the abend is requested by the assembler user exit or whether the ABTERMENC(ABEND) runtime option is used.

Abend codes generated by CEEBXITA

When you request an abend through CEEBXITA, the values contained in certain fields of the exit are used for the abend code and the reason code. Table 27 on page 136 shows the abend codes used by Language Environment when CEEBXITA requests an abend and does not modify the CEEAUE_RETC code field.

Table 27: Abend codes used by Language Environment when the Assembler user exit requests an abend

<table>
<thead>
<tr>
<th>Condition severity</th>
<th>User return code</th>
<th>Abend code in non-CICS</th>
<th>Abend code in CICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>User abend 2000</td>
<td>Transaction abend 2000</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>User abend 3000</td>
<td>Transaction abend 3000</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>User abend 4000</td>
<td>Transaction abend 4000</td>
</tr>
</tbody>
</table>

Abend codes generated by ABTERMENC(ABEND) runtime option

Language Environment terminates the enclave with the same abend code that caused the unhandled condition of severity 2 or greater if all of the following are true:

- You use the ABTERMENC(ABEND) runtime option.
- The unhandled condition was generated by an abend.
- The assembler user exit does not alter the CEEAUE_ABND flag setting.

Table 28 on page 136 shows the abend code and reason code used when the enclave terminates due to the various unhandled conditions of severity 2 or greater and ABTERMENC(ABEND) is specified in both CICS and non-CICS environments.

Table 28: Abend code values used by Language Environment with ABTERMENC(ABEND).

<table>
<thead>
<tr>
<th>Unhandled condition</th>
<th>Abend code</th>
<th>Abend reason code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABEND</td>
<td>The original abend code</td>
<td>In non-CICS environment, the original abend reason code</td>
</tr>
</tbody>
</table>
### Table 28: Abend code values used by Language Environment with ABTERMENC(ABEND). (continued)

<table>
<thead>
<tr>
<th>Unhandled condition</th>
<th>Abend code</th>
<th>Abend reason code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program interrupt</strong></td>
<td>See “Program interrupt abend and reason codes” on page 137 for program interrupt abend codes</td>
<td></td>
</tr>
<tr>
<td><strong>Software-raised condition</strong></td>
<td>A user 4038 abend is used in a non-CICS environment and a transaction 4038 abend is used in a CICS environment</td>
<td>In a non-CICS environment, X'1'</td>
</tr>
<tr>
<td><strong>Unsuccessful LOAD (non-CICS)</strong></td>
<td>The abend code that would have been used by the operating system.</td>
<td>The abend reason code that would have been used by the operating system.</td>
</tr>
</tbody>
</table>

**Note:** In a CICS environment, when an abend is issued, only the abend code is returned. CICS does not return an abend reason code.

### Program interrupt abend and reason codes

A program interrupt can cause an unhandled condition of severity 2 or greater. When running with the ABTERMENC(ABEND) runtime option in a CICS environment, an abend code of ASRA is issued for program interrupts. When running with the ABTERMENC(ABEND) runtime option in a non-CICS environment, the abend codes and reason codes shown in Table 29 on page 137 are issued for program interrupts.

### Table 29: Program interrupt abend and reason codes in a non-CICS environment.

<table>
<thead>
<tr>
<th>Program interrupts</th>
<th>Abend code</th>
<th>Abend reason code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation exception</td>
<td>SOC1</td>
<td>00000001</td>
</tr>
<tr>
<td>Privileged operation exception</td>
<td>SOC2</td>
<td>00000002</td>
</tr>
<tr>
<td>Execute exception</td>
<td>SOC3</td>
<td>00000003</td>
</tr>
<tr>
<td>Protection exception</td>
<td>SOC4</td>
<td>00000004</td>
</tr>
<tr>
<td>Segment translation exception (note 1)</td>
<td>SOC4</td>
<td>00000004</td>
</tr>
<tr>
<td>Page translation exception (note 2)</td>
<td>SOC4</td>
<td>00000004</td>
</tr>
<tr>
<td>Addressing exception</td>
<td>SOC5</td>
<td>00000005</td>
</tr>
<tr>
<td>Specification exception</td>
<td>SOC6</td>
<td>00000006</td>
</tr>
<tr>
<td>Data exception</td>
<td>SOC7</td>
<td>00000007</td>
</tr>
<tr>
<td>Fixed-point overflow exception</td>
<td>SOC8</td>
<td>00000008</td>
</tr>
<tr>
<td>Fixed-point divide exception</td>
<td>SOC9</td>
<td>00000009</td>
</tr>
<tr>
<td>Decimal overflow exception</td>
<td>SOCA</td>
<td>0000000A</td>
</tr>
<tr>
<td>Decimal divide exception</td>
<td>SOCB</td>
<td>0000000B</td>
</tr>
<tr>
<td>Exponent overflow exception</td>
<td>SOCC</td>
<td>0000000C</td>
</tr>
<tr>
<td>Exponent underflow exception</td>
<td>SOCD</td>
<td>0000000D</td>
</tr>
<tr>
<td>Significance exception</td>
<td>SOCE</td>
<td>0000000E</td>
</tr>
<tr>
<td>Floating-point divide exception</td>
<td>SOCF</td>
<td>0000000F</td>
</tr>
</tbody>
</table>

**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 13. Program management model

Now that you have been introduced to how applications run in Language Environment, you need to understand the model of program management under which Language Environment operates. Understanding the model helps you recognize equivalent entities across Language Environment-conforming programming languages and predict how your single- and mixed-language applications run. This topic provides an overview of the Language Environment model.

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

The POSIX program management model differs somewhat from the Language Environment program management model. Refer to “Mapping the POSIX program management model to the Language Environment program management model” on page 143 for more information.

The Language Environment program management model has three basic entities — the process, enclave, and thread, each of which Language Environment creates whenever you start execution of an HLL application. A description is provided of the entities and their relationship to program management.

Model terminology for Language Environment program management

Some terms used to describe the program management 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, see 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 management 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.

Equivalent HLL terms: C or C++ – program, consisting of a main C or C++ function and its subroutines, COBOL – run unit, Fortran – executable program, consisting of a main Fortran program and its subprograms, PL/I – main procedure and all its subprocedures.

Thread
An execution entity that consists of synchronous invocations and terminations of routines. The thread is the basic runtime path within the Language Environment program management model; dispatched by the system with its own runtime stack, instruction counter, and registers.

Routine
In Language Environment, either a procedure, function, or subroutine.

Equivalent HLL terms: C or C++ – function, COBOL – program, Fortran – program, PL/I – procedure, BEGIN/END block.
Program management model

## Terminology for data

### Automatic data
Data that does not persist across calls. In the absence of a specific 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++, or Fortran, any non-EXTERNAL data items in COBOL, and data with the PL/I INTERNAL attribute (whether implicitly, or by explicit declaration).

Figure 51 on page 140 shows the simplest form of the Language Environment program management model and the resources that each component controls. Refer to the figure as you read about the program management model.
**Processes**

A process is a collection of resources, both application code and data, consisting of one or more related enclaves. The process is the outermost or highest level runtime component of the common runtime 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 runtime environment.

**Enclaves**

A key feature of the program management 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, just as a COBOL run unit defines the scope of semantics of a COBOL program. Scope encompasses names, external data sharing, and control statements such as C’s `exit()`, COBOL’s STOP RUN, Fortran’s STOP, and PL/I’s STOP and EXIT statements.

**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 PL/I application would be the PROC OPTIONS (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, for example, data declared using the PL/I INIT() option. 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 external storage class, COBOL data items defined with the EXTERNAL attribute, Fortran common blocks, and PL/I data declared as EXTERNAL.

- 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 or PL/I variable with block scope, any Fortran data declared as AUTOMATIC, and any non-EXTERNAL data item in COBOL.

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()`, COBOL's STOP RUN, Fortran's STOP and END statements, and PL/I's STOP and EXIT statements are examples of such statements. When one of these statements is executed, the main routine within the enclave terminates. Thus, the enclave defines the scope of the language statements.

Before returning, resources obtained by the routines in the enclave are released and any open files (other than the Language Environment message file) are closed.

Additional enclave characteristics

Management of resources
The enclave manages most Language Environment resources, such as the thread and heap storage, other than the message file (which is managed as a process-level resource). 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 is shared between enclaves.

Multiple enclaves
z/OS Language Environment provides explicit support for a single enclave within a single process. Under some circumstances, however, multiple enclaves can exist within a single process. A description of how to create multiple, or nested, enclaves can be found in Chapter 31, “Using nested enclaves,” on page 469.

Threads
Within each enclave is a thread, the basic runtime path represented by the machine state; conditions raised during execution are isolated to that runtime 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 runtime stack, instruction counter, registers, and condition handling mechanisms.

Because threads operate with unique runtime 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 management model
Figure 52 on page 143 illustrates the relationship between the various entities that make up the Language Environment program management model.

As Figure 52 on page 143 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 is the enclave. The threads can create enclaves, which can create more threads, and so on.
Mapping the POSIX program management model to the Language Environment program management 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 management model which differs somewhat from the Language Environment program management model. This section provides a helpful comparison of both models.

The descriptions 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.

A Language Environment process with multiple enclaves is a Language Environment extension to POSIX. If a process contains more than one enclave, only the first enclave in the process can have POSIX(ON) specified. All of the nested enclaves must be POSIX(OFF). A process that contains any POSIX(OFF) enclaves cannot issue a `fork()` or a `spawn()`, either explicitly or implicitly (`popen()` being mapped to `fork()` and `exec()`).

**Note:** The scope of a specific POSIX function might be the Language Environment process or Language Environment enclave. See “Scope of POSIX semantics” on page 144 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 bounded 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 53 on page 145:

- 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.
For a Language Environment process with a single enclave, scoping to the process or to the (only) enclave has the same semantic effect.

Figure 53: Scope of semantics against POSIX processes and Language Environment processes and enclaves
Chapter 14. 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.

How Language Environment-conforming languages uses stack and heap storage

<table>
<thead>
<tr>
<th>Language</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>C or C++</td>
<td>• Automatic variables</td>
<td>Variables allocated by:</td>
</tr>
<tr>
<td></td>
<td>• Library routines</td>
<td>• malloc() function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• calloc() function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• realloc() function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Static external (RENT)</td>
</tr>
<tr>
<td>COBOL</td>
<td>• Intrinsic functions</td>
<td>• WORKING-STORAGE variables</td>
</tr>
<tr>
<td></td>
<td>• Library routines</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• LOCAL-STORAGE variables</td>
<td></td>
</tr>
<tr>
<td>Fortran</td>
<td>• Library routines</td>
<td>• Dynamic common blocks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Variables allocated by ALLOCATE statement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(VS Fortran Version 2 Release 6 only)</td>
</tr>
<tr>
<td>PL/I</td>
<td>• Automatic variables</td>
<td>• BASED variables</td>
</tr>
<tr>
<td></td>
<td>• Library routines</td>
<td>• CONTROLLED variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• AREA variables</td>
</tr>
</tbody>
</table>

Related runtime options

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANYHEAP</td>
<td>Allocates library (HLL and Language Environment) heap storage above or</td>
</tr>
<tr>
<td></td>
<td>below 16 MB</td>
</tr>
<tr>
<td>BELOWHEAP</td>
<td>Allocates library heap storage below 16 MB</td>
</tr>
<tr>
<td>HEAP</td>
<td>Allocates storage for user-controlled dynamically allocated variables</td>
</tr>
<tr>
<td>HEAPCHK</td>
<td>Specifies that heap storage be inspected for damage.</td>
</tr>
<tr>
<td>HEAPPOOLS</td>
<td>Improves the performance of heap storage allocation</td>
</tr>
<tr>
<td>HEAPZONES</td>
<td>Provides a heap check zone for each storage request</td>
</tr>
</tbody>
</table>
### Table 31: Runtime options and functions (continued)

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBSTACK</td>
<td>On non-CICS, used by library routine stack frames that must be below 16 MB</td>
</tr>
<tr>
<td>RPTSTG</td>
<td>Generates a storage report</td>
</tr>
<tr>
<td>STACK</td>
<td>Used by library routine stack frames that can reside anywhere in storage</td>
</tr>
<tr>
<td>STORAGE</td>
<td>Controls the initial content and amount of storage reserved for the out-of-storage condition</td>
</tr>
<tr>
<td>THREADHEAP</td>
<td>Controls the allocation and management of thread-level heap storage</td>
</tr>
<tr>
<td>THREADSTACK</td>
<td>Controls the upward- and downward-growing stack allocation for each thread, except the initial thread, in a multithreaded environment</td>
</tr>
</tbody>
</table>


### Related callable services

#### Related callable services:

**Table 32: Callable services options and functions**

<table>
<thead>
<tr>
<th>Callable service</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEECRHP</td>
<td>Defines additional heaps</td>
</tr>
<tr>
<td>CEECZST</td>
<td>Changes the size of a previously allocated heap element</td>
</tr>
<tr>
<td>CEEDSHP</td>
<td>Discards an entire heap created with CEECRHP</td>
</tr>
<tr>
<td>CEEFRST</td>
<td>Frees storage allocated by CEEGTST or an intrinsic language function</td>
</tr>
<tr>
<td>CEEGTST</td>
<td>Gets storage from a heap whose ID you specify</td>
</tr>
<tr>
<td>CEE3RPH</td>
<td>Sets the heading displayed at the top of the storage options report</td>
</tr>
</tbody>
</table>


### Stack storage overview

**Note:** The term stack refers to the user stack, which is an independent area of stack storage that can be located above or below the 16 MB line, designed to be used by both library routines and compiled code. All references to stack storage and stack frame 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 allocated on entry to a routine or block, and freed on the subsequent return. It is a contiguous area of storage obtained directly from the operating system. Stack storage is automatically provided at thread initialization and is available in the user stack.

The user stack is used by both library routines and, except for Fortran, compiled code. Stack storage is also available in the library stack, which is an independent area of stack storage, allocated below the 16 MB line, designed to be used only by library routines.

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 large segments of storage called stack segments, which are further 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 first segment used for stack storage is called the initial stack segment. When the initial stack segment becomes full, a second segment, or stack increment is obtained from the operating system. As each succeeding stack increment becomes full, another is obtained from the operating system as needed. The size of the initial stack segment and the size of the increments are specified by the init_size and incr_size parameters of the STACK runtime option. For more information about the STACK runtime option, see z/OS Language Environment Programming Reference.

Figure 54 on page 149 shows the standard Language Environment stack storage model. The XPLINK stack (see Figure 17 on page 28) is structured differently. See Chapter 3, “Using Extra Performance Linkage (XPLINK),” on page 25 for information about XPLINK.

**Tuning stack storage**

For best performance, the initial stack segment should be large enough to satisfy all requests for stack storage. The Language Environment storage report generated by the RPTSTG(ON) option (see z/OS Language Environment Programming Reference) shows you how much stack storage is being used, the total number of segments allocated to the stack, and the recommended values for the STACK runtime option. An initial stack segment that is too large can waste storage and degrade overall system performance, especially under CICS where storage is limited.

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

RPTSTG(ON) and the STORAGE runtime option (see z/OS Language Environment Programming Reference) 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.
COBOL storage considerations

Storage for data items declared in the COBOL LOCAL-STORAGE SECTION is allocated from the Language Environment user stack. The storage location of data items declared in COBOL LOCAL-STORAGE SECTION is controlled by the Language Environment STACK option. The COBOL compiler options do not affect the location of data items declared in the COBOL LOCAL-STORAGE SECTION.

PL/I storage considerations

PL/I automatic storage is provided by the Language Environment user stack. Automatic storage above the 16 MB line is supported under control of the Language Environment STACK and THREADSTACK runtime options. When the Language Environment user stack is above 16 MB, PL/I temporaries (dummy arguments) and parameter lists (for reentrant/recursive blocks) also reside above 16 MB. As long as an OS PL/I application does not contain edited stream I/O (for example, the EDIT option is not used in a PUT statement) and is running with AMODE(31), you can relink it with Language Environment to allow for STACK(,,ANY) to be used. For Enterprise PL/I for z/OS and PL/I for MVS & VM, as long as the application is AMODE(31), STACK(,,ANY) is supported. The stack frame size for an individual block is constrained to 16 MB, which means the size of an automatic aggregate, temporary variable, or dummy argument cannot exceed 16 MB.

Heap storage overview

Heap storage is used to allocate storage that has a lifetime not related to the execution of the current routine; it remains allocated until you explicitly free it or until the enclave terminates. You can control allocation and freeing of heap storage using Language Environment callable services, and tune heap storage using the Language Environment runtime options HEAP, THREADHEAP and HEAPPOOLS; see z/OS Language Environment Programming Reference for details.

Heap storage is shared among all program units and all threads in an enclave. Any thread can free heap storage. You can free one element at a time with the CEEFRST callable service, or you can free all heap elements at once using CEEDSHP. You cannot, however, discard the initial heap.

Storage can be allocated or freed with any of the HLL storage facilities, such as malloc(), calloc(), or ALLOCATE, along with the Language Environment storage services. For HLLs with no intrinsic function for storage management, such as COBOL, you can use the Language Environment storage services.

When HEAPPOOLS(ON) or HEAPPOOLS(ALIGN) is in effect, the C storage management intrinsic functions must be used together. That is, if you malloc(), you must use free() to release the storage, you cannot use CEEFRST. See “Using HEAPPOOLS to improve performance” on page 151 for more information about heap pools.

Heap storage, sometimes referred to as a heap, is a collection of one or more heap segments comprised of an initial heap segment, which is dynamically allocated at the first request for heap storage, and, as needed, one or more heap increments, allocated as additional storage is required. The initial heap is provided by Language Environment and does not require a call to the CEECRHP service. The initial heap is identified by heap_id=0. It is also known as the user heap. See Figure 55 on page 151 for an illustration of Language Environment heap storage.

Heap segments, which are contiguous areas of storage obtained directly from the operating system, are subdivided into individual heap elements. Heap elements are obtained by a call to the CEEGTST service, and are allocated within each segment of the initial heap by the Language Environment storage management routines. When the initial heap segment becomes full, Language Environment gets another segment, or increment, from the operating system.

The size of the initial heap segment is governed by the init_size parameter of the HEAP runtime option. (See z/OS Language Environment Programming Reference.) The incr_size parameter governs the size of each heap increment.

A named heap is set up specifically by a call to the CEECRHP service, which returns an identifier when the heap is created. Additional heaps can also be created and controlled by calls to CEECRHP.
Additional heaps provide isolation between logical groups of data in different additional heaps. Separate additional heaps when you need to group storage objects together so they can be freed at once (with a single call to CEEDSHP), rather than freed one element at a time (with calls to CEEFRST).

Library routines occasionally use a heap called the library heap for storage below 16 MB. The size of this heap is controlled by the BELOWHEAP runtime option. The library heap and the BELOWHEAP runtime option have no relation to heaps created by CEECRHP. If an application program creates a heap using CEECRHP, library routines never use that heap (except, of course, the storage management library routines CEEGTST, CEEFRST, CEECZST, and CEEDSHP). The library heap can be tuned with the BELOWHEAP runtime option.

The Language Environment anywhere heap and below heap are reserved for runtime library usage only. Application data and variables are not kept in these heaps. You normally should not adjust the size of these heaps unless the storage report indicates excessive segments allocated for the anywhere or below heaps, or if too much storage has been allocated.

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

Language Environment provides a memory leak analysis tool (MEMCHECK) to perform the following functions:

- Check for heap storage leaks, double frees, overlays and print them in a report.
- Trace user heap storage allocation and deallocation requests and print them in a report.

See z/OS Language Environment Debugging Guide for more information on MEMCHECK.

You can use the HEAPCHK runtime option to run heap storage tests and to help identify storage leaks. The HEAPZONES runtime option can be used to identify storage overlay damage.


Figure 55 on page 151 shows the Language Environment heap storage model.

Using HEAPPOOLS 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. The sizes of the cells, the number of pools for each size and cell pool extents are specified by the HEAPPOOLS runtime option, which is also used to enable the heap pools algorithm.

**Note:** Use of the Vendor Heap Manager (VHM) overrides the use of the HEAPPOOLS runtime option.

## Applications that 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 65536 bytes or less, because the heap pools algorithm is not used in a `malloc()` that is greater than 65536 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 percentages are carefully tuned. (For tuning information, see “Tuning heap storage” on page 153.) It is possible that some applications running with the heap pools algorithm will have to increase their region size.

## Heap pools modes of operation

Heap pools can be operated in two modes:

- **ON** This mode is selected by specifying the runtime option HEAPPOOLS(ON). This mode will avoid 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

Contestion 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 portion of the threads will be allocating from each pool. For most cell sizes, there is little contention and one pool is 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 can 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.

## Heap IDs recognized by the Language Environment heap manager

Table 33 on page 152 lists Language Environment heaps and their respective purposes.

<table>
<thead>
<tr>
<th>Heap name</th>
<th>Heap ID</th>
<th>Intended purpose</th>
<th>Created by</th>
<th>Disposed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial heap user heap</td>
<td>0</td>
<td>Application program data. Common heap used by language intrinsic functions and COBOL WORKING-STOREAGE data items. CEEDSHIP has no effect on the initial heap. COBOL access is by Language Environment callable services.</td>
<td>Enclave initialization. Size and location determined from HEAP runtime option.</td>
<td>Enclave termination</td>
</tr>
</tbody>
</table>
Table 33: Heap IDs recognized by Language Environment heap manager (continued)

<table>
<thead>
<tr>
<th>Heap name</th>
<th>Heap ID</th>
<th>Intended purpose</th>
<th>Created by</th>
<th>Disposed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additional heaps and user heap</td>
<td>(Returned by CEECRHP)</td>
<td>Collections of application program data that can be quickly disposed with a single CEEDSHP call.</td>
<td>Call to CEECRHP. Arguments define heap size, location, and other characteristics.</td>
<td>Call to CEEDSHP Enclave termination</td>
</tr>
</tbody>
</table>

AMODE considerations for heap storage

The `initsz24` and `incrsz24` parameters of the HEAP runtime option control the initial size and subsequent increments of heap storage allocated below the 16 MB line. This storage is required for AMODE(24) applications running with the ALL31(OFF) and HEAP(,,ANYWHERE) runtime options in effect.

For example, suppose the initial heap segment is allocated above 16 MB. If an AMODE(24) routine requests storage from this initial heap, Language Environment must allocate a heap segment from below the 16 MB line so that the AMODE(24) routine can address the storage.

When a Fortran program is in AMODE(24), heap storage is allocated below the 16 MB line. The allocation of heap storage in a Fortran common block is sensitive to the AMODE setting of the requester program. For example, if a requester in AMODE(31) calls a Fortran program in AMODE(24), heap storage is allocated above the 16 MB line as defined by the AMODE(31) setting of the requester.

OS PL/I uses HEAP(,,ANYWHERE) as the default location for heap storage. The allocation of heap storage is sensitive to the AMODE setting of the requester and the main program. If the requester is in AMODE(31) or HEAP(,,ANYWHERE) is in effect and the main program is in AMODE(31), heap storage is allocated above the 16 MB line.

There are some restrictions when using CEEGTST, the Get Heap Storage AWI, in an AMODE(24) COBOL program. See the description of CEEGTST in z/OS Language Environment Programming Reference for details on these restrictions.

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 (see z/OS Language Environment Programming Reference) 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 HEAP, ANYHEAP, BELOWHEAP and HEAPPOLS runtime options. For PL/I multitasking applications, the Language Environment THREADHEAP runtime option can be used to tune heap storage at the task level.

The heap pools algorithm (see “Using HEAPPOLS to improve performance” on page 151) 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 heap pools algorithm for an application is a three-step process:

1. Run your application with the runtime options HEAPPOLS(ON) or HEAPPOLS(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 HEAPPOLS 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 HEAPPOLS option. Examine the storage report.

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 (see z/OS Language Environment Programming Reference) 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.

Usage notes:
1. These recommendations are dynamic and represent values for this particular run. The values might change with each run performed.
2. Long-running applications might have an adverse effect on the statistical data collection. Fixed length counters might overflow, causing incorrect HEAPPOOLS recommendations. If the recommendations appear to be unrealistic, rerun with a reduced application run time.

Storage performance considerations

Use the RPTSTG(ON) option to generate a report about the amount of storage your application uses in various Language Environment storage classes (such as STACK, THREADSTACK, LIBSTACK, THREADHEAP, HEAP, HEAPPOOLS and BELOWHEAP). You can also use the report to determine your application's minimum storage requirements and the number of segments allocated and freed, and the manner in which heap pool cells are being used. You can use this information to tune your application to minimize the number of segments allocated and freed, and to increase the efficiency of the heap pools algorithm. Before putting your application into production, be sure to specify the RPTSTG(OFF) option so that no storage report is generated. RPTSTG(ON) can have a negative affect on the performance of your application, because as the application runs, statistics are kept on storage requests.

Dynamic storage services

Language Environment provides callable services that let you get and free heap storage at selected points in your application. Stack storage is automatically allocated upon entry into a routine and freed upon exit, but you must allocate heap storage, which persists until you free it or until your application terminates.

Each time your application runs, the setting of the HEAP runtime option specifies the size of an initial heap from which heap storage is allocated. You can allocate storage out of this initial heap whenever your application requires it. Call CEEGTST (Get Heap Storage) and specify an ID identifying the initial heap and the portion of storage in the initial heap that you require. When your application no longer requires the storage, you can call the CEEFRST (Free Heap Storage) service with the address of the element to free it. CEECRHP (Create New Additional Heap) allows you to identify a heap, other than the initial heap, from which to get and free storage. You can use CEEGTST to allocate elements from the newly created heap. One advantage of this approach is that CEECRHP allows you to group storage elements together and to use CEEDSHP (Discard Heap) to discard them all at once when you no longer need them.

For a description and syntax of each Language Environment dynamic storage callable service, see z/OS Language Environment Programming Reference.

Callable services are not supported directly from a Fortran program. For information about invoking callable services from assembler routines, see Chapter 29, “Assembler considerations,” on page 391.

Examples of callable storage services

This topic contains examples that use callable services. The first group of examples use CEEGTST and CEEFRST to build a linked list. The second group of examples use CEE3RHP, CEECRHP, CEEGTST, CEECZST, CEEFRST, and CEEDSHP to manage storage.
C example of building a linked list

Following is an example of how to build a linked list in a C program using callable services.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

void main () {
    // Module/ File Name: EDCLLST
    _INT4 HEAPID;
    _INT4 HPSIZE;
    _INT4 LCOUNT;
    _FEEDBACK FC;
    _POINTER ADDRSS;
    struct LIST_ITEM {
        _INT4 COUNT;
        struct LIST_ITEM *NEXT_ITEM;
    };
    struct LIST_ITEM *ANCHOR;
    struct LIST_ITEM *CURRENT;
    _INT4 NBYTES = sizeof(struct LIST_ITEM);
    printf ( "\n******************************************************\n" );
    printf ( "CESCSTO C Example is now in motion\n" );
    printf ( "\n******************************************************\n" );
    ANCHOR = NULL;
    for ( LCOUNT = 1; LCOUNT < 4; LCOUNT++ )
    { /* Call CEEGTST to get storage from user heap */
        CEEGTST ( &HEAPID , &NBYTES , &ADDRSS , &FC );
        if ( (_FBCHECK (FC , CEE000) == 0) && ADDRSS != 0 )
            /* If storage is gotten successfully, the linked */
            /* list element is initialized and appended to the */
            /* list. The list origin is pointed to by the * */
            /* variable ANCHOR. */
            { if (ANCHOR == NULL)
                // Stack and heap storage
                ANCHOR =(struct LIST_ITEM *) ADDRSS;
            }else{
                CURRENT -> NEXT_ITEM =(struct LIST_ITEM *)ADDRSS;
                CURRENT =(struct LIST_ITEM *) ADDRSS;
                CURRENT -> NEXT_ITEM = NULL;
                CURRENT -> COUNT = LCOUNT;
            }
            printf ( "\nError in getting user storage\n" );
            // Stack and heap storage
        }
    }
}
```
COBOL example of building a linked list

Following is an example of how to build a linked list in a COBOL program using callable services.

CBL C,LIB,RENT,LIST,QUOTE
*Module/File Name: IGZTLLST
****************************************************************
**                                                             *
** CESCSTO - Drive CEEGTST - obtain storage from user heap      *
**           and CEEFRST - free linked list storage              *
**                                                             *
** This example illustrates the construction of a linked       *
** list using the LE storage management services.              *
**                                                             *
** 1. Storage for each list element is allocated from the      *
**    user heap,                                               *
** 2. The list element is initialized and appended to the      *
**    list.                                                   *
** 3. After three members are appended, the list traversed     *
**    and the data saved in each element is displayed.         *
** 4. The linklist storage is freed.                          *
**                                                             *
****************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. CESCSTO.
DATA DIVISION.
****************************************************************
** Storage management parameters, including pointers **
** for the returned storage addresses. **
****************************************************************
WORKING-STORAGE SECTION.
01 LCOUNT PIC 9 USAGE DISPLAY VALUE 0.
01  HEAPID       PIC S9(9) BINARY VALUE 0.
01  NBYTES      PIC S9(9) BINARY.
01  FC.
   05  FILLER    PIC X(8).
   COPY CEEIGZCT.
   05  FILLER    PIC X(4).
01  ADDRSS USAGE IS POINTER VALUE NULL.
01  ANCHOR USAGE IS POINTER VALUE NULL.

*******************************************************
** Define variables in linkage section in order to **
** reference storage returned as addresses in         **
** pointer variables by Language Environment.        **
*******************************************************
LINKAGE SECTION.
01  LIST-ITEM.
   05  CHARDATA   PIC X(80) USAGE DISPLAY.
   05  NEXT-ITEM USAGE IS  POINTER.
PROCEDURE DIVISION.
0001-BEGIN-PROCESSING.
   DISPLAY "***************************************".
   DISPLAY "CESCSTO COBOL Example is now in motion.".
   DISPLAY "***************************************".
   *******************************************************
   ** Call CEEGTST to get storage from user heap       **
   *******************************************************
   MOVE LENGTH OF LIST-ITEM TO NBYTES
   PERFORM 3 TIMES
      ADD 1 TO LCOUNT
      CALL "CEEGTST" USING HEAPID , NBYTES,
      ADDRSS , FC
   *******************************************************
   ** If storage storage is gotten successfully, an     **
   ** address is returned by Le in the ADDRSS           **
   ** parameter. The address of variable LIST-ITEM      **
   ** in the linkage section can now be SET to address  **
   ** the acquired storage. LIST-ITEM is appended to    **
   ** the end of the list. The list origin is pointed   **
   ** to by the variable ANCHOR.                       **
   *******************************************************
   IF CEE000 THEN
      IF ANCHOR = NULL THEN
         SET ANCHOR TO ADDRSS
      ELSE
         SET NEXT-ITEM TO ADDRSS
      END-IF
      SET ADDRESS OF LIST-ITEM TO ADDRSS
      SET NEXT-ITEM TO NULL
      MOVE " " TO CHARDATA
      STRING "This is list item number " LCOUNT
         DELIMITED BY SIZE INTO CHARDATA
      ELSE
         DISPLAY "Error in obtaining storage from heap"
         GOBACK
      END-IF
   ELSE
      DISPLAY "Error in obtaining storage from heap"
      GOBACK
   END-IF
   END-PERFORM.
   *******************************************************
   ** On completion of the above loop, we have the      **
   ** following layout:                                 **
   ** ANCHOR --> LIST-ITEM1 --> LIST-ITEM2 --> LIST-ITEM3 **
   ** Loop thru list items 1 thru 3 and print out the   **
   ** identifying text written in the CHARDATA fields.  **
   ** Test a counter variable to verify that three items **
   ** were indeed in the linked list.                   **
   *******************************************************
   MOVE 0 TO LCOUNT.
   PERFORM WITH TEST AFTER UNTIL (ANCHOR = NULL)
      SET ADDRESS OF LIST-ITEM TO ANCHOR
      DISPLAY CHARDATA
      SET ADDRESS TO ANCHOR
      SET ANCHOR TO NEXT-ITEM
      PERFORM 100-FREESTOR
      ADD 1 TO LCOUNT
   END-PERFORM.
   IF (LCOUNT = 3 ) THEN
      DISPLAY "****************************".
      DISPLAY "CESCSTO COBOL Example is now ended.  "
      DISPLAY "****************************"
   ELSE
   END-IF

Stack and heap storage
PL/I example of building a linked list

Following is an example of how to build a linked list in an Enterprise PL/I for z/OS or PL/I for MVS & VM program using callable services.

```pli
*PROCESS MACRO;
*Process lc(101),opt(0),s,map,stmt,a(f),ag         ;
CE CSCSTO: PROC OPTIONS (MAIN);
%INCLUDE  CEEIBMAM;
%INCLUDE  CEEIBMCT;
/*********************************************************
/*Module/File Name: IBMLLST                          */
/*********************************************************
/**                                                **/
/** FUNCTION : CEEGTST - obtain storage from user  **/
/**                      heap for a linked list.   **/
/**          : CEEFRST - free linked list storage  **/
/**                                                **/
/**   This example illustrates the construction of **/
/**   a linked list using the Language Environment **/
/**   storage management services.                 **/
/**                                                **/
/**   1. Storage for each list element is         **/
/**       allocated from the user heap,            **/
/**                                                **/
/**   2. The list element is initialized and      **/
/**       appended to the list.                    **/
/**                                                **/
/**   3. After three members are appended, the    **/
/**       list traversed and the data saved in     **/
/**       each element is displayed.               **/
/**                                                **/
/**   4. The linklist storage is freed.          **/
/**                                                **/
/*********************************************************
DCL NULL BUILTIN;
/*********************************************************
/*  Declare linked list item as based structure.    */
/*********************************************************
DCL 01 LIST_ITEM BASED(ADDRSS), /* Map of list item */
    02  CHARDATA CHAR(80),
    02  NEXT_ITEM POINTER;
**
DCL HEAPID FIXED BIN(31,0) INIT (0); /* heap ID for user heap */
DCL NBYTES FIXED BIN(31,0) /* size of required heap */
   INIT (STORAGE(LIST_ITEM));
DCL 01 FC, /* Feedback token */
   03 MsgSev REAL FIXED BINARY(15,8),
   03 MsgNo REAL FIXED BINARY(15,8),
   03 Flags,
      05 Case BIT(2),
      05 Severity BIT(3),
      05 Control BIT(3),
   03 FacID CHAR(3), /* Facility ID */
   03 ISI /* Instance-Specific Information */
      REAL FIXED BINARY(31,0);
DCL ADDRSS POINTER, /* Address of storage */
  PREV POINTER; /* Address of prior item */
DCL ANCHOR POINTER; /* linklist anchor */
/*********************************************************
/* Declare linked list item as based structure.       */
/*********************************************************
DCL 01 LIST_ITEM BASED(ADDRSS), /* Map of list item */
   02  CHARDATA CHAR(80),
   02  NEXT_ITEM POINTER;
```
C example of storage management

Following is an example of how to manage storage for a C program using callable services.

/*Module/File Name: EDCSTOR*/
/*Function : CEE3RPH - Set report heading*/
//****
This example illustrates the invocation of the Language Environment Dynamic Storage Callable Services for a C program.

1. A report heading is set for display at the beginning of the storage or options report.
2. A user heap is created.
3. Storage is allocated from the user heap.
4. A change is made to the size of the allocated storage.
5. The allocated storage is freed.
6. The user heap is discarded.

---

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

void main ()
{
    _CHAR80 RPTHEAD;
    _INT4 HEAPID;
    _INT4 HPSIZE;
    _INT4 NBYTES;
    _INT4 INCR;
    _INT4 OPTS;
    _INT4 STORALC;
    _POINTER ADDRSS;
    _FEEDBACK FC;

    printf ( "\n**********************************\n" );
    printf ( "CE90STO C Example is now in motion\n" );
    printf ( "\n**********************************\n" );
    memset ( RPTHEAD , ' ' , 80 );
    memcpy ( RPTHEAD , "User defined report heading" , 27 );
    CEE3RPH ( RPTHEAD , &FC );
    if ( _FBCHECK ( FC , CEE000 ) != 0 )
        printf ( "Error in setting report heading\n" );

    CEECRHP ( &HEAPID , &HPSIZE , &INCR , &OPTS , &FC );
    if ( _FBCHECK ( FC , CEE000 ) != 0 )
    {
        printf ( "\n*********************************************\n" );
        printf ( "Call CEE3RPH to set the user defined report heading*\n*********************************************\n" );
        CEE3RPH ( RPTHEAD , &FC );
        if ( _FBCHECK ( FC , CEE000 ) != 0 )
            printf ( "Error in setting report heading\n" );
        printf ( "Call CEECRHP to create a user heap*\n*********************************************\n" );
    }

    HEAPID = 0;
    HPSIZE = 1;
    INCR = 0;
    OPTS = 0;
    STORALC = 0;
    CEECRHP ( &HEAPID , &HPSIZE , &INCR , &OPTS , &FC );
    if ( _FBCHECK ( FC , CEE000 ) != 0 )
    {
        printf ( "\n*********************************************\n" );
        printf ( "Call CEEGTST to get storage from user heap*\n*********************************************\n" );
        NBYTES = 4000;
        CEEGTST ( &HEAPID , &NBYTES , &ADDRSS , &FC );
        if ( ( _FBCHECK ( FC , CEE000 ) != 0 ) && ADDRSS != 0 )
            printf ( "Error in getting user storage\n" );
        printf ( "Call CEECZST to change size of heap element*\n*********************************************\n" );
        NBYTES = 2000;
        CEECZST ( &ADDRSS , &NBYTES , &FC );
        if ( ( _FBCHECK ( FC , CEE000 ) != 0 )
        {
            STORALC = 1;
        }else{
            printf ( "Error in changing size of storage\n" );
        }
        printf ( "Error in getting user storage\n" );
    }
}
```
COBOL example of storage management

Following is an example of how to manage storage for a COBOL program using callable services.

CBL LIB,QUOTE
*Module/File Name: IGZTSTOR
*******************************************************************************
** CE90STO - Call the following LE services:
**   : CEE3RPH - Set report heading
**   : CEECRHP - Create user heap
**   : CEEGTST - obtain storage from user heap
**   : CEECZST - change size of this piece of storage
**   : CEEFRST - free this piece of storage
**   : CEEDSHP - discard user heap
**   This example illustrates the invocation of the LE
**   Dynamic Storage Callable Services from a COBOL program.
**   1. A report heading is set for display at the beginning
**      of the storage or options report.
**   2. A user heap is created.
**   3. Storage is allocated from the user heap.
**   4. A change is made to the size of the allocated storage.
**   5. The allocated storage is freed.
**   6. The user heap is discarded.
*******************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. CE90STO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 RPTHEAD PIC X(80).
01 HEAPID PIC S9(9) BINARY.
01 HPSIZE PIC S9(9) BINARY.
01 INCR PIC S9(9) BINARY.
01 OPTS PIC S9(9) BINARY.
01 ADDRS USAGE IS POINTER.
01 NBYTES PIC S9(9) BINARY.
01 NEWSIZE PIC S9(9) BINARY.
01 FC.
  02 Condition-Token-Value.
  COPY CEEIGZCT.
    03 Case-1-Condition-ID.
      04 Severity PIC S9(4) BINARY.
      04 Msg-No PIC S9(4) BINARY.
    03 Case-2-Condition-ID
      REDEFINES Case-1-Condition-ID.
      04 Class-Code PIC S9(4) BINARY.
      04 Cause-Code PIC S9(4) BINARY.
    03 Case-Seq-Ctl PIC X.
    03 Facility-ID PIC XXX.
  02 I-S-Info PIC S9(9) BINARY.
PL/I example of storage management

Following is an example of how to manage storage for an Enterprise PL/I for z/OS or a PL/I for MVS & VM program using callable services.

*PROCESS MACRO;
CE90STO: PROC OPTIONS(MAIN);

%INCLUDE  CEEIBMAW;
%INCLUDE  CEEIBMCT;
/******************************************************/
This example illustrates the use of the Language Environment storage callable services in a PL/I program.

1. A report heading is set for display at the beginning of the storage or options report.
2. A user heap is created.
3. Storage is allocated from the user heap.
4. The size of allocated storage is changed.
5. The allocated storage is freed.
6. The user heap is discarded.

Stack and heap 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++ Runtime 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 for non-XPLINK and XPLINK.

The VHM will not manage the following:

- `CEEQTST`
• CEEVGTST
• CEEFRST
• CEEVFRST
• CEECZST
• CEEVCZST
• CEEVGTSB
• Additional heaps (CEECRHP)
• User-created heaps (__ucreate, __umalloc, __ufree)
• ANYHEAP
• BELOWHEAP

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 environment variable
• Inside the file specified by the _CEE_ENVFILE_S environment variable. _CEE_ENVFILE_S will enable a list of environment variables to be set from a specified file and will also strip trailing white space from each name=value line read from a file.

You must set the environment variable before any user code obtains control; that is, before the HLL user exit, static constructors, or main obtains control. If you set it after the user code has begun executing, the VHM will not be activated but the value of the environment variable will be updated.
Chapter 15. Introduction to Language Environment condition handling

This section 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.

If you use mixed-language applications, it is especially important for you to know how Language Environment condition handling works with existing high-level language (HLL) condition handling schemes.

Described in detail later in this section are the steps involved in condition handling under Language Environment, HLL-specific condition handling considerations, Language Environment—POSIX signal handling interactions, and how you can communicate events that happen in a routine to another routine.

If your application is running under CICS, you should refer to the CICS-specific condition handling information, which is discussed in “Condition handling under CICS” on page 359. If your application is running under IMS, you should refer to the IMS-specific condition handling information, which is discussed in “Condition handling under IMS” on page 369.

Concepts of Language Environment condition handling

There are two main concepts of Language Environment condition handling: the stack frame-based model and the unique, 12-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 condition 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 12-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.

Language Environment supplements, but does not replace, existing HLL condition handling techniques such as C/C++ signal handlers (created using the signal() function), PL/I ON-units, and return code-based programming techniques. HLL condition handling techniques are discussed in Chapter 16, “Language Environment and HLL condition handling interactions,” on page 183.

Language Environment condition handling is most beneficial when used as part of mixed-language applications because it is consistent for all applications. If you are coding in a single language, you can use the condition handling semantics of that language, but if you have any ILC applications, you need the consistency across languages that Language Environment provides.

Language Environment can respond in many ways to a condition. For example, Language Environment can invoke a condition handler, a term used to define the specific routine that actually recognizes and responds to the condition. A condition handler can be registered by the CEEHDLR (register user-written condition handler) service, or be part of the language-specific condition handling services, such as a C/C++ signal handler or a PL/I ON-unit. HLL condition handling semantics that are intrinsic to the programming language also exist; an example is the COBOL ON SIZE phrase.

Related runtime options are as follows:
Condition handling introduction

ABPERC
Percolates (removes from Language Environment condition handling) a single abend

DEPTCHONDLMT
Indicates how deep conditions might be nested

ERRCOUNT
Indicates how many severity 2, 3, and 4 conditions can occur before issuing an abend.

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

XUFLOW
Indicates if exponent underflow should cause program interrupt.

Related callable services are as follows:

CEE3CIB
Returns pointer to the condition information block that is associated with a condition token passed to a user-written condition handler

CEE3GRN
Gets name of routine that incurred the condition currently being processed

CEE3GRO
Returns the offset of the location within the most current Language Environment-conforming routine where a condition occurred

CEE3SPM
Queries or modifies (by enabling or masking) hardware conditions

CEE3SRP
Sets a resume point within user application code to resume from a Language Environment user condition handler

CEEGQDT
Retrieves q_data token from the ISI

CEEHDLR
Registers user-written condition handler

CEEHDLU
Unregisters user-written condition handler

CEEITOK
Returns the initial condition token from the current condition information block

CEEMRCE
Moves the resume cursor to an explicit location where resumption is to occur after a condition has been handled

CEEMRCC
Moves resume cursor relative to handle cursor. You might view this as performing a GOTO out of block, or setjmp() and longjmp().

CEESGL
Signals a condition

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. 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 distinction between global condition handling and condition handling within a stack frame-based model can affect how a condition is handled in your application, particularly if it is a mixed-language 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
- Entry into a program in COBOL
- Entry into a main program or subprogram in Fortran
- Entry into a procedure or begin block in PL/I
- Entry into an ON-unit in PL/I

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.

Each new stack frame can contain user-written condition handlers registered with CEEHDLR, but language-specific handlers such as C/C++ signal handlers are not associated with each stack frame. User condition handlers can be unregistered explicitly (by calling CEEHDLU) or implicitly, as when the routine that registered the handler returns control to its caller.

Two cursors, or pointers, keep track of the state of condition handling. The cursors are named the handle and resume cursors.

Handle cursor

If a condition occurs or is raised, the handle cursor initially points to the most recently established condition handler within the stack frame. As condition handling progresses, the handle cursor moves to earlier handlers within the stack frame, or to the first handler in the calling stack frame.

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. You can move the resume cursor relative to the handle cursor by calling CEEMRCR. You can use CEEMRCE to move the resume cursor to an explicit location in the application when the application resumes.

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 (using CEESGL), language library routines, language constructs (such as C's `raise()` or PL/I's `SIGNAL`), or user-written condition handling routines.

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” on page 170 and in detail in Chapter 16, “Language Environment and HLL condition handling interactions,” on page 183), Language Environment might deem an exception to be a condition, at which point it can be handled by Language Environment, user-written condition 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 imminent steps.

During the condition and termination imminent steps, the stack is used to determine the order of condition handler processing. Condition handlers associated with the most recent stack frame added to the stack are given first chance to handle the condition. Condition handlers associated with the next stack frame are next given a chance, and so on until either the condition is handled or default Language Environment condition handling semantics take effect.

In a POSIX(OFF) environment, only routines that are currently active on the stack have an effect on condition handling. For example, in a COBOL — PL/I application, a COBOL main program calls a PL/I subroutine. The subroutine then returns control to COBOL. The PL/I routine is no longer on the stack and does not affect condition handling. See “Language Environment and POSIX signal handling interactions” on page 199 for information about signal handling under z/OS UNIX.

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 later in this section, the default enablement of your HLL applies.

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” on page 171 for more information.

• Specify in the assembler user exit or ABPERC runtime option an abend code or list of codes to be percolated (passed to the operating system). See “Language Environment abends and the enablement step” on page 171 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 or a PL/I `NOZERODIVIDE` prefix in a PL/I procedure 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 handlers never get a chance to handle the program check because it is not considered a condition.
  – Call the CEE3SPM callable service or use the XUFLOW runtime option to disable hardware conditions.
In summary, not all hardware interrupts, software conditions, or user-signaled events become conditions. Those that are not ignored and do become conditions enter the condition step. See “Condition step” on page 172 for the details of what takes place during the condition step.

**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, C/C++ signal handlers, PL/I ON-units, and user-written condition handlers can then be invoked to handle them. An exception to this behavior is that Language Environment cannot handle Sx22 abends, even if TRAP(ON) is specified.

**CEESGL and TRAP**

When a condition is raised using the CEESGL callable service, C/C++ signal handlers, PL/I ON-units, and user-written condition handlers are always invoked if present, regardless of the setting of TRAP. If none of these handle the condition, then HLL semantics’ default action could be taken. See z/OS Language Environment Programming Reference for more information about TRAP and CEESGL.

**Language Environment abends and the enablement step**

You can prevent Language Environment from automatically issuing abends for certain exceptions by requesting that an abend code or codes be percolated. If an abend is percolated, neither Language Environment nor an HLL can handle it; only the operating system can respond to the abend.

Abends that are not retryable (for example, x37 ABENDs) are always percolated. Additionally, abends can be percolated in three ways:

- You can specify in the assembler user exit CEEXBITA a list of abend codes that Language Environment percolates. You can specify both system abends and user abends. See Chapter 28, “Using runtime user exits,” on page 373 for more information.

- The ABPERC runtime option allows you to specify which (if any) abend code should be percolated by Language Environment. ABPERC is intended for use as a debugging tool that allows the application to execute with TRAP(ON).

  For a list of Language Environment-issued abends and information about using ABPERC to debug your application, see z/OS Language Environment Runtime Messages and z/OS Language Environment Debugging Guide. See z/OS Language Environment Programming Reference for more information about the TRAP and ABPERC runtime options.

- If an abend is issued from a request block (RB) on which Language Environment did not establish an ESTAE, the abend will be percolated to the system.

  For example, COBOL is the main program, running on an RB, where Language Environment establishes an ESTAE. The COBOL application determines an error condition has occurred, and it calls an assembler program. This assembler program issues an SVC LINK to a non-Language Environment enabled assembler program, creating a new RB. Neither of these assembler programs establishes an ESTAE or an ESPIE. The assembler program on the new RB, where no Language Environment recovery has been established, issues an SVC ABEND. When the SVC ABEND is issued from the second RB, Language Environment, which is only active on the first RB, will percolate the abend to the system.

  As a result, Language Environment member languages will not be called for termination processing and any open files may be closed by MVS task termination. If there are DCB exits associated with any of the open files, these exits may be called from task termination, and may result in some unexpected S0Cx abends since the member language did not initiate the close of the files.

  To avoid these scenarios, do one of the following:
- Make the assembler program Language Environment conforming by using CEEENTRY and CEETERM macros.
- Do not use SVC LINK to issue abends from assembler programs. Use the Language Environment callable service CEE3ABD.
- Make sure that all files are closed in the application before linking to an assembler routine to issue an abend.

Using CEE3SPM and XUFLOW to enable and disable hardware conditions

You can change the enablement of certain hardware interrupts using the CEE3SPM callable service and XUFLOW runtime option under Language Environment (see z/OS Language Environment Programming Reference for syntax).

Language Environment provides the CEE3SPM callable service to replace assembler language routines that manipulate bits 20 through 23 of the Program Status Word (PSW) to enable or disable the following hardware interrupts:

- Decimal overflow
- Exponent underflow
- Fixed-point overflow
- Significance

The XUFLOW runtime option specifies whether an exponent underflow exception causes a program interrupt. Both CEE3SPM and XUFLOW can change the condition handling semantics of the HLL or HLLs of your application. Therefore, use CEE3SPM and XUFLOW only if you understand the effect they have on your application.

C and C++ considerations

C and C++ ignore requests to enable the decimal overflow, exponent underflow, fixed-point overflow, or significance exceptions.

COBOL considerations

The decimal overflow and fixed-point overflow exceptions cannot be enabled in a COBOL program; COBOL ignores any request to enable these exceptions.

Fortran considerations

The fixed-point overflow, decimal overflow, and exponent underflow masks are ON by default. Mask settings remain in effect until changed by CEE3SPM or XUFLOW, or until the application calls a new load module containing code from a language that specifies the masks ON.

The Fortran XUFLOW callable service can affect the semantics of any ILC application or any program setting made with CEE3SPM.

PL/I considerations

PL/I semantics depend on the program mask being given certain settings:

- The fixed-point overflow, decimal overflow, and exponent underflow masks are ON. For Enterprise PL/I for z/OS, the fixed-point overflow mask is OFF.
- The significance mask is OFF.

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. Condition handlers are invoked at each intervening stack frame and given a chance to respond in any of the ways described in “Responses to conditions” on page 178. The condition step lasts until a condition handler requests a resume or until default condition handling occurs (condition went unhandled). Throughout the following discussion, refer to Figure 56 on page 173.

Figure 56: 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 56 on page 173, this is A, or routine 4.

2. If the debug tool is present, and the setting of the TEST runtime option indicates that it should be given control, it is invoked. See z/OS Language Environment Programming Reference for information about the TEST runtime option.

3. If the debug tool is not invoked, or does not handle the condition, Language Environment traverses the stack, stack frame by stack frame, towards earlier stack frames. This is in the direction of arrow B in Figure 56 on page 173. User-written condition handlers established using CEEHDLR, and then language-specific condition handlers present at each stack frame such as C/C++ signal handlers or PL/I ON-units, can all respond by percolating, promoting, or handling the condition (see “Responses to conditions” on page 178 for a discussion of these actions).

4. 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.

5. If the HLL of the routine that originated the condition does not issue a resume, what occurs next depends on whether there is a PL/I routine active on the stack.

   a. The condition is percolated if there is no currently active PL/I routine or if the condition is not one that PL/I promotes to the ERROR condition (see “Promoting conditions to the PL/I ERROR condition” on page 196 for details). Language Environment default actions are then taken based on the severity of the unhandled condition, as indicated in Table 34 on page 174.

      If the condition is of severity 2 or above, Language Environment promotes the condition to T_I_U (termination imminent due to an unhandled condition) and returns to routine 4 to redrive the stack (this occurs at points C and D in Figure 56 on page 173). For more information about the termination imminent step and T_I_U, see “Termination imminent step” on page 175.

   b. If the condition is one that PL/I promotes to the PL/I ERROR condition (see “Promoting conditions to the PL/I ERROR condition” on page 196 for details), the condition is promoted at the location represented as C in Figure 56 on page 173, and another pass is made of the stack. The following takes place:

      • On the next pass of the stack (D), any ERROR ON-unit or user-written condition handler is invoked. If the ON-unit or user-written condition handler issues a resume, condition handling ends. Execution resumes where the resume cursor points.
• If no ON-unit or user-written condition handler issues a resume, the ERROR condition is promoted (at E) to T_I_U. (See “Processing the T_I_U condition” on page 175 for a discussion of T_I_U.)

• A final pass of the stack is made, beginning in Routine 4 where the original condition occurred (F). Because T_I_U maps to the PL/I FINISH condition, both established PL/I FINISH ON-units and user-written condition handlers registered for T_I_U are invoked.

• If no user-written or HLL condition handlers act on the condition, Language Environment begins thread termination activities in response to the unhandled condition (G). See Table 34 on page 174 for the default actions that Language Environment takes for conditions of different severity levels.

Influencing condition handling with the ERRCOUNT runtime option

The ERRCOUNT option allows you to specify the number of errors that are tolerated during the execution of a thread. Each condition of severity 2 or above, regardless of its origin, increments the error count by one. POSIX conditions are not counted. If the error count exceeds the limit, Language Environment terminates the enclave with abend code 4091 and reason code 11.

See z/OS Language Environment Programming Reference for syntax and more information about using ERRCOUNT.

<table>
<thead>
<tr>
<th>Severity of condition</th>
<th>Condition signaled by user in a call to CEESGL with an fc</th>
<th>Condition came from any other source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (informative message)</td>
<td>Return CEE069 condition token, and resume processing at the next sequential instruction.</td>
<td>Resume without issuing message.</td>
</tr>
<tr>
<td></td>
<td>See the fc table for CEESGL (z/OS Language Environment Programming Reference) for a description of the CEE069 condition token.</td>
<td></td>
</tr>
<tr>
<td>1 (warning message)</td>
<td>Return CEE069 condition token, and resume processing at the next sequential instruction.</td>
<td>If the condition occurred in a stack frame associated with a COBOL program, resume and issue the message.</td>
</tr>
<tr>
<td></td>
<td>If the condition occurred in a stack frame associated with a non-COBOL routine, resume without issuing message.</td>
<td></td>
</tr>
<tr>
<td>2 (program terminated in error)</td>
<td>Return CEE069 condition token, and resume processing at the next sequential instruction.</td>
<td>Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.</td>
</tr>
<tr>
<td></td>
<td>See “Processing the T_I_U condition” on page 175 for more information about T_I_U.</td>
<td></td>
</tr>
</tbody>
</table>
Default responses to unhandled conditions. The 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. (continued)

<table>
<thead>
<tr>
<th>Severity of condition</th>
<th>Condition signaled by user in a call to CEESGL with an fc</th>
<th>Condition came from any other source</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 (Program terminated in severe error)</td>
<td>Return CEE069 condition token, and resume processing at the next sequential instruction.</td>
<td>Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.</td>
</tr>
<tr>
<td>4 (program terminated in critical error)</td>
<td>Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.</td>
<td>Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.</td>
</tr>
</tbody>
</table>

Termination imminent step

The termination imminent step occurs for certain unhandled conditions or as the result of STOP-like language constructs such as C/C++ `exit()` or `abort();` Fortran STOP statement or a call to EXIT, SYSRCX, or DUMP; or PL/I STOP. The termination imminent step occurs when one of the following events occurs:

- The T_I_U condition (Termination Imminent due to Unhandled condition) is raised
- The T_I_S condition (Termination Imminent due to Stop) is raised to indicate that the thread can potentially terminate

When T_I_U or T_I_S is raised, another pass is made of the stack. See “Processing the T_I_U condition” on page 175 and “Processing the T_I_S condition” on page 176 for details on what can happen during and after the pass.

You can directly signal T_I_U and T_I_S using the CEESGL callable service. When you do, Language Environment behaves as described in “CEESGL and the termination imminent step” on page 177.

Processing the T_I_U condition

Table 34 on page 174 indicates that for severity 4 conditions signaled by CEESGL, and for severity 2 and above conditions that remain unhandled after all condition handlers have had a chance to handle them, Language Environment promotes the unhandled condition to T_I_U. See z/OS Language Environment Programming Reference for a discussion of CEESGL. T_I_U is a severity 3 condition with the representation shown in Table 35 on page 175:

<table>
<thead>
<tr>
<th>Symbolic feedback code (fc)</th>
<th>Severity</th>
<th>Message number</th>
<th>Message text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE066</td>
<td>3</td>
<td>0198</td>
<td>Termination of a thread was signaled.</td>
</tr>
</tbody>
</table>

After promoting the condition to T_I_U, Language Environment does the following:

1. Language Environment revisits each stack frame on the stack, beginning with the stack frame in which the condition occurred, and progressing towards earlier stack frames. At each stack frame, HLL and user-written condition handlers are given a chance to handle the condition.

The T_I_U condition maps to the PL/I FINISH condition. Therefore, an established PL/I FINISH ON-unit or registered user-written condition handler can be invoked to handle the condition. After the ON-
unit or condition handler completes its processing, the termination activities described in Step 3 take place.

2. If, during the course of condition handling, the resume cursor is moved and a resume is requested by a condition handler, execution resumes at the instruction pointed to by the resume cursor. If a resume is requested for the T_I_U condition without moving the resume cursor, the thread terminates immediately with no clean-up. See z/OS Language Environment Programming Reference for a discussion of the CEEMRRC service.

3. If all stack frames have been visited, and the condition remains unhandled, or a FINISH ON-unit or user-written condition handler has processed the condition and returned, Language Environment performs the following termination activities:
   - Sets the reason and return codes. The return code value is based on the severity of the original unhandled condition, not on the T_I_U condition (which is a severity 3).
   - Issues a message for the condition.
   - Prints a traceback and dump depending on the setting of the TERMTHDACT runtime option (see z/OS Language Environment Programming Reference for syntax)
   - Terminates the thread.

In this release of Language Environment, multithreading is supported only in a POSIX(ON) environment. Unless your application is doing multithreading, therefore, when a thread terminates, the entire enclave terminates.

Processing the T_I_S condition

The termination imminent step of condition handling can also be entered as the result of the T_I_S (Termination_Imminent due to STOP) condition being signaled. T_I_S is a severity 1 condition with the following representation:

<table>
<thead>
<tr>
<th>Symbolic feedback code (fc)</th>
<th>Severity</th>
<th>Message number</th>
<th>Message text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE067</td>
<td>1</td>
<td>0199</td>
<td>Termination of a thread was signaled.</td>
</tr>
</tbody>
</table>

The T_I_S condition is raised by Language Environment immediately upon detection of a language STOP-like construct such as:
   - C/C++ exit() function
   - COBOL STOP RUN
   - Fortran STOP statement
   - Fortran END statement in a main program
   - PL/I EXIT statement
   - PL/I STOP statement

The HLL constructs listed above initiate termination activities for the enclave in two steps:

1. Language Environment traverses the stack beginning at the stack frame for the routine containing the STOP-like statement and proceeds, stack frame by stack frame, towards earlier stack frames. User-written and HLL condition handlers at each stack frame are given a chance to handle the condition.

   T_I_S maps to the PL/I FINISH condition. Therefore, both established PL/I FINISH ON-units and user-written condition handlers can be invoked. After the ON-unit or condition handler completes its processing, the termination activities described in Step 2 take place.

2. If all stack frames have been visited, and the condition remains unhandled, or an ON-unit or condition handler has processed the condition and returned, Language Environment:
- Sets the reason and return codes
- Terminates the thread

Language Environment performs only one pass of the stack for STOP-like statements.

**Termination imminent step and the TERMTTHDACT runtime option**

You can use the TERMTTHDACT 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.

TERMTTHDACT behavior under z/OS UNIX differs slightly; for details, see “Termination imminent step under z/OS UNIX” on page 202.

**PL/I considerations**

For those PL/I conditions that do not raise the ERROR condition as part of their implicit action, PL/I requires that a message be issued. For these conditions, the message is issued regardless of the setting of TERMTTHDACT. Therefore, messages can be delivered even when TERMTTHDACT(QUIET) is set.

If the condition remains unhandled (for example, the PL/I FINISH condition is still regarded as unhandled after normal return from a FINISH ON-unit), and the application terminates, the message associated with the condition is not issued again at termination.

For more information about TERMTTHDACT, see z/OS Language Environment Debugging Guide and z/OS Language Environment Programming Reference.

**CEESGL and the termination imminent step**

You can signal T_I_U and T_I_S directly with the CEESGL callable service. Two reasons you might need to do this are:

- To force the driving of a FINISH ON-unit or similar construct that would perform clean-up activities
- To test a PL/I ON-unit or user-written condition handler that you have designed to handle T_I_U or T_I_S

If you signal T_I_U or T_I_S by calling CEESGL with the feedback code parameter, the following occurs:

1. Language Environment visits each stack frame on the stack, beginning with the stack frame in which the condition was signaled, and progressing towards older stack frames. At each stack frame, HLL and user-written condition handlers are given a chance to handle the condition.

   T_I_U and T_I_S both map to the PL/I FINISH condition. Therefore, an established PL/I FINISH ON-unit can be invoked to handle the condition.

2. If all stack frames have been visited, and the condition remains unhandled, or a FINISH ON-unit has processed the condition and returned, Language Environment returns the CEE069 condition token to the routine that called CEESGL, and processing resumes at the next sequential instruction.

**Invoking condition 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 debug tool, a user-written condition handler, or a language-specific condition handling mechanism:

**z/OS Debugger**

If you have invoked a debug tool using the TEST runtime option or the CEETEST callable service (see z/OS Language Environment Programming Reference), the debug tool gains control when a condition occurs. Unless a condition is promoted and is passed through the stack again for additional condition handling, a debug tool is invoked only once per stack.
User-written condition handler

User-written condition handlers are routines that you supply to handle specific conditions that might arise in the runtime environment. As shown in Figure 57 on page 178, a LIFO queue containing zero or more user-written condition handlers is associated with each stack frame. A different queue exists for each stack frame. For example, if routine A calls routine B, there is a new queue associated with the stack frame for routine B.

```
+-----------------+        +-----------------+
| Routine A       |        | Handler Routine A1 |
| Register A1     |        | Handler Routine A2 |
| Register A2     |        |                    |
| Invoke Routine B|        |                    |
```

Figure 57: Queues of user-written condition handlers

User-written condition handlers are registered on a stack frame-by-stack frame basis using the CEEHDLR callable service. A call to CEEHDLR from a given routine adds a user-written condition handler onto the queue for the stack frame associated with that routine. Registering a condition handling routine using CEEHDLR implicitly requests Language Environment to pass control to this routine when a condition occurs. For example, you could call CEEHDLR to register two user-written condition handlers for the same stack frame, one that handles floating-point underflow conditions and another that handles floating-point divide conditions.

The most recent user condition handler registered using CEEHDLR is the first to be invoked by Language Environment. Note that you could also register a single user condition handler to handle both of these conditions.

The user-written condition handlers can respond to a condition in any of the ways described in “Responses to conditions” on page 178.

User-written condition handlers are given a chance to handle a given condition before the language-specific condition handling semantics take effect.

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

Condition 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. The resume cursor can be manipulated to be placed at a specific point by using the CEEMRCR (move resume cursor) callable service (see z/OS Language Environment Programming Reference).

**Percolate**
A condition is percolated if a condition handler declines to handle it. User-written condition handlers, for example, can be written to act on a particular condition, but percolate all other conditions. Language Environment can continue condition handling in one of the following places:

- 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 condition handlers, or the language-specific condition semantics.
- With the most recently established condition handler associated with the calling stack frame.

**Promote**
A condition is promoted when a condition handler converts the condition into one with a different meaning. A condition handler can promote a condition for a variety of reasons, including the condition handler's knowledge or lack of knowledge about the cause of the original condition. A condition can be promoted to simulate conditions that would normally come from a different source.

**Fix-up and resume**
The qualifying data is modified and a resume occurs with a corrective action. There are several possible responses that can be applied:

- **Resume with new input value**
  A new input value is specified and the failing operation is tried again. The condition token for this action has the condition name CEE0CE.

- **Resume with new output value**
  The program continues using a specified result in the place of what the failing operation would have provided. The condition token for this action has the condition name CEE0CF.

For more information about how these responses can be used in developing user-written condition handlers, see “User-written condition handler interface” on page 204.

---

**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 16, “Language Environment and HLL condition handling interactions,” on page 183 if you are interested in specific HLL condition handling behavior.

**Scenario 1: Simple condition handling**

Refer to Figure 58 on page 180 throughout the following discussion.
In this scenario, no C/C++ handlers created by a call to `signal()`, PL/I ON-units, or user-written condition handlers registered using the CEEHDLR service are established at any stack frame in the application.

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 any user-written condition handlers have been registered using the CEEHDLR callable service on the routine B's stack frame, they are given control. No handlers have been registered, so the condition is percolated.
   - If a C/C++ signal handler is registered, or if a PL/I ON-unit is established on the stack frame, it is given control. Neither one exists on routine B's stack frame, so the condition is percolated.
   - If any user-written condition handlers have been registered using CEEHDLR on routine A's stack frame, they are given control. No handlers have been registered, so the condition is percolated.
   - If a C/C++ signal handler is registered or if a PL/I ON-unit is established on routine A's stack frame, it is given control. No C/C++ signal handler or PL/I ON-unit has been established for the stack frame, 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), promotes the condition to T_I_U, and requests that the stack be redriven. This is the end of the condition step and the beginning of the termination imminent step.
4. The following occurs during the termination imminent step:
   - The stack frame for routine B is revisited, and if a user-written condition handler is present, it is given control. No handlers are registered, so T_I_U is percolated.
   - If a C/C++ signal handler or PL/I ON-unit can respond to the T_I_U condition, it is given control. In this case, there are none, so the condition is percolated.
   - The stack frame for routine A is revisited, and checked for user-written condition handlers registered for the T_I_U condition, C/C++ signal handlers or PL/I ON-units. No handlers are registered, so T_I_U is percolated.
   - Language Environment takes the default action for the unhandled T_I_U condition, which terminates the enclave.
Scenario 2: User-written condition handler present for T_I_U

Scenario 2 is much the same as Scenario 1, except that routine A does have a user-written condition handler established. Refer to Figure 59 on page 181 throughout the following scenario.

In this scenario, routine A is a routine that invokes other prewritten applications. If any of the components of the prewritten application fail, routine A must remain up and take alternate action. Therefore, routine A has a user-written condition handler registered. The handler is designed to handle the T_I_U condition by issuing a nonlocal jump to a location within routine A. The handler percolates all conditions other than T_I_U.

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 condition handler has been registered for the divide-by-zero condition on routine B's stack frame, it is given control. One has not been registered, so the condition is percolated.
   - If a C/C++ signal handler has been registered or a PL/I ON-unit has been established for the divide-by-zero condition, it is given control. No C/C++ signal handler or ON-unit is present, so the condition is percolated to Language Environment.
   - If a user-written condition handler has been registered on routine A's stack frame, it is given control. However, because the divide-by-zero condition is not the one the handler is looking for, the condition is percolated.
   - If a C/C++ signal handler is registered or a PL/I ON-unit is established for the condition on routine A's stack frame, it is given control. Neither one is present, so the condition is percolated.
   - After the earliest stack frame (in this case, that for routine A) has been checked, HLL and Language Environment default actions occur. In this case, assume that the HLL percolates the condition to Language Environment.

Language Environment examines the severity of the unhandled divide-by-zero condition (severity 3), promotes the condition to T_I_U, and requests that the stack be redriven. This is the end of the condition step and the beginning of the termination imminent step.

4. The following occurs during the termination imminent step:
   - Language Environment revisits the stack frame for routine B, checking for user-written condition handlers registered for the T_I_U condition. No handlers are registered, so T_I_U is percolated.
   - If a PL/I FINISH ON-unit is present, it is given control. In this example, there isn't one, so the condition is percolated.
   - Language Environment revisits the stack frames for routine A, checking for user-written condition handlers registered for the T_I_U condition. There is one, it is given control. The user code in the
handler, using either HLL or Language Environment facilities, causes control to pass to a location within routine A.

5. Control resumes with routine A at the location specified. The condition is now handled.

**Scenario 3: Condition handler present for divide-by-zero**

Scenario 3 is much the same as scenario 2, except that routine B has a user-written condition handler established to handle the divide-by-zero condition. Refer to Figure 60 on page 182 throughout the following scenario.

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 condition handler has been registered using the CEEHDLR callable service 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 causes resumption either through HLL constructs or Language Environment services.
4. The condition is now considered to be handled and is never seen by stack frame A or the Language Environment default handler.

![Figure 60: Scenario 3: Division by zero with a user handler present in routine B](image-url)
Chapter 16. Language Environment and HLL condition handling interactions

This section discusses the condition handling discussion. It would be helpful for you to read Chapter 15, “Introduction to Language Environment condition handling,” on page 167 before reading this topic. Chapter 15, “Introduction to Language Environment condition handling,” on page 167 introduces you to terminology and concepts that are discussed in the present topic, 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.

This topic discusses HLL condition handling semantics, focusing on how HLL semantics interact with the Language Environment condition handling model and services. C, C++, COBOL, Fortran, and PL/I are each discussed, and condition handling scenarios and examples are provided. This topic 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” on page 183
- “C++ condition handling semantics” on page 190
- “COBOL condition handling semantics” on page 191
- “Fortran condition handling semantics” on page 194
- “PL/I condition handling semantics” on page 195
- “Language Environment and POSIX signal handling interactions” on page 199

If you are running a single-language application written in C or PL/I, 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.

However, if you are running a single-language application written in COBOL or assembler that has little built-in error handling, you might notice a change in how errors are handled under Language Environment. For example, in an application that relies on abend codes to handle errors, you might need to alter the assembler user exit to get the same behavior under Language Environment as under the previous runtime environment. See Chapter 28, “Using runtime user exits,” on page 373 for information about modifying the assembler user exit.

For information about condition handling in ILC applications, see z/OS Language Environment Writing Interlanguage Communication Applications.

C condition handling semantics

This section describes C condition handling in a POSIX(OFF) environment. If you run applications that contain POSIX functions, you should also read “Language Environment and POSIX signal handling interactions” on page 199, 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 37 on page 184 contains default C-language error handling semantics.
### Table 37: C conditions and default system actions

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

### 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 Language Environment services, you register a condition handler by using CEEHDLR, and you raise a condition by using CEESGL.

• 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. C++-unique exception handling functions are discussed in “C++ condition handling semantics” on page 190.

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 37 on page 184. 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.

  The signal handler specified in signal() is given a chance to handle a condition only after any user-written handler established using CEEHDLR is invoked.

Using the raise() function

When the C raise() function is called for any of the conditions listed in Table 37 on page 184, a corresponding Language Environment condition is automatically raised by a call to the CEESGL callable service. Any of these conditions (EDC6000 through EDC6004) can be handled by a user-written condition handler registered using the CEEHDLR service. For detailed descriptions of conditions EDC6000 through EDC6004, see z/OS XL C/C++ Programming Guide and z/OS Language Environment Runtime Messages.

For more information about the CEEHDLR and CEESGL callable services, see z/OS Language Environment Programming Reference. For more information about using the raise() function, see z/OS XL C/C++ Runtime Library Reference.

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 condition 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 or PL/I STOP or EXIT statement 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.

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 37 on page 184.
     - You did not specify any action, but the default action for the condition is SIG_IGN (see Table 37 on page 184).
     - You are running under CICS and a CICS handler is pending.
   - 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 the debug tool is present, and the setting of the TEST runtime option indicates that it should be given control, it is invoked. See z/OS Language Environment Programming Reference for information about the TEST runtime option.
     - If the debug tool is not invoked, or does not handle the condition, any user-written condition handlers registered using CEEHDLR for that stack frame are invoked.
     - If no user-written condition handlers are registered for the condition that has occurred, and 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 you do, the condition is percolated to the next user-written condition handler registered using CEEHDLR, or to the language-specific condition handler associated with the next stack frame.
   - If condition 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 34 on page 174 take place.
• If the Language Environment default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition), the termination imminent step of condition handling is entered.

3. When the condition is promoted to T_I_U, Language Environment makes another pass of the stack looking for user-written condition handlers registered for T_I_U.

If, on the next pass of the stack, no condition handler issued a resume or moved the resume cursor, 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 condition handlers registered.

There is no user-written condition handler or 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. A message, trace, Language Environment dump, or a user address space dump could be generated depending on the setting of TERMTHDACT (see z/OS Language Environment Programming Reference).

Figure 61 on page 187 is a C main routine that calls C370B, a subroutine that passes data to another subroutine, C370C.

```c
#include <stdio.h>
int y = 0;
void C370B(void);
int main(void) {
    printf("In Program C370A\n");
    C370B();
}
```

Figure 61: C370A routine

Figure 62 on page 188 is a C subroutine that calls C370C, and passes data to it.
/*Module/File Name: EDCMLTB */
万物dìng 发 dà xià de yí lù jìng dà fāng xiàng dào C370C. */
/* C370C wèi qin cì yí chà yì wù zháo duì fàn. */
/**************************************************************************/

#include <stdio.h>
extern int y;
void C370C(int);
void C370B(void) {
  int x;
  printf("In Program C370B\n");
  x = y;
  C370C(x);
}

Figure 62: C370B routine

Figure 63 on page 188 generates a divide-by-zero. The divide-by-zero condition is percolated back to
C370B, to C370A, and to Language Environment default behavior.

/*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 63: C370C routine

Condition occurs with signal handler present

Figure 64 on page 189 contains a simple example of a C application in which $y = \frac{a}{b}$ 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.
If b = 0, a floating-point divide condition occurs. Language Environment condition handling begins:

- The enablement step occurs.
  - If Table 37 on page 184 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 a debug tool is present, it receives control.
- If a user-written condition handler is registered by CEEHDLR for that stack frame, it receives control.

If none of the above takes place, the condition manager gives the C signal-handler control. This handler in turn invokes \texttt{c_handler()} as specified in the \texttt{signal()} function in Figure 64 on page 189. 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 condition 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)
- You do not request in the assembler user exit or in the ABPERC runtime option that any of the abends be percolated (ABPERC(NONE))

Following are the C signal representations for the following exceptions.
• For S/370 exceptions generated by the hardware or math library, see Table 38 on page 190. Some of the exceptions listed in the table can be masked off for normal Language Environment execution.

• For abends, see Table 39 on page 190.

Table 38: Mapping of S/370 exceptions to C signals

<table>
<thead>
<tr>
<th>Interrupt code</th>
<th>Interrupt code description</th>
<th>C signal type</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Operation exception</td>
<td>SIGILL</td>
</tr>
<tr>
<td>02</td>
<td>Privileged-operation exception</td>
<td>SIGILL</td>
</tr>
<tr>
<td>03</td>
<td>Execution exception</td>
<td>SIGILL</td>
</tr>
<tr>
<td>04</td>
<td>Protection exception</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>05</td>
<td>Addressing exception</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>06</td>
<td>Specification exception</td>
<td>SIGILL</td>
</tr>
<tr>
<td>07</td>
<td>Data exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>08</td>
<td>Fixed-point overflow exception</td>
<td>n/a</td>
</tr>
<tr>
<td>09</td>
<td>Fixed-point divide exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>10</td>
<td>Decimal-overflow exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>11</td>
<td>Decimal-divide exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>12</td>
<td>Exponent-overflow exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>13</td>
<td>Exponent-underflow exception</td>
<td>n/a</td>
</tr>
<tr>
<td>14</td>
<td>Significance exception</td>
<td>n/a</td>
</tr>
<tr>
<td>15</td>
<td>Floating-point divide exception</td>
<td>SIGFPE</td>
</tr>
</tbody>
</table>

Table 39 on page 190 lists the C signal type for abends that can occur under Language Environment.

Table 39: Mapping of abend signals to C signals

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

**C++ condition handling semantics**

C++ includes the C condition handling model and new C++ constructs `throw`, `try`, and `catch`. For more information about 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 183. 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, COBOL, PL/I, or Language Environment exceptions.

## COBOL condition handling semantics

COBOL native condition handling is very different from C, PL/I, or Fortran native condition handling.

COBOL provides some condition handling on a statement-by-statement basis; for example, the ON EXCEPTION phrase of the CALL statement, the ON EXCEPTION phrase of the INVOKE statement, and the ON SIZE ERROR phrase of the COMPUTE statement. For other conditions, COBOL generally reports the error. An assembler user exit is available for COBOL to specify events that should cause an abend.

For more information about user exits, see Chapter 28, “Using runtime user exits,” on page 373. For a discussion of COBOL condition handling in an ILC application, see z/OS Language Environment Writing Interlanguage Communication Applications. The following discussion applies to stacks comprised solely of COBOL programs.

If an exception occurs in a COBOL program, COBOL does nothing until every condition handler at every stack frame has been interrogated.

After all stack frames have been visited, COBOL does the following:

1. Checks to see if the condition has a facility ID of IGZ (is a COBOL-specific condition). If not, COBOL percolates the condition to the Language Environment condition manager.
2. Handles the condition based on its severity (see Table 34 on page 174 for an explanation of severity codes and their meaning under Language Environment).
   - If the condition severity is 1, a message describing the condition is issued to the destination specified in the MSGFILE runtime option, and processing resumes in the program in which the error occurred.
   - If the severity is 2 or above, COBOL percolates the condition to the Language Environment condition manager. The Language Environment default action then takes place.

## COBOL condition handling examples

The following examples demonstrate how conditions are handled in Language Environment if you do not use any Language Environment callable services, and do not have any user-written condition handlers registered. The COBOLA program in Figure 65 on page 192 calls COBOLB in Figure 66 on page 192, which in turn calls the COBOLC program, in Figure 67 on page 193. A divide-by-zero condition occurs in COBOLC.

The divide-by-zero is enabled as a condition, so the condition step of Language Environment condition handling is entered. There is no user-written condition handler that is registered for COBOLC or any of the other COBOL programs, so the condition is percolated through all of the stack frames. COBOL's default action for the divide-by-zero condition is to percolate the condition to Language Environment. The divide-by-zero condition has a severity of 3. The Language Environment default response to an unhandled severity 3 condition is to terminate the application and issue a message if TERMTHDACT(MSG) is specified.
Demonstrate a failing COBOL program with multiple active routines on the stack. The call sequence is as follows: COBOLA --> COBOLB --> COBOLC (which causes a zero divide).

**Figure 65: COBOLA program**

Figure 66 on page 192 calls COBOLC and passes data to it.

Second routine called in the following call sequence: COBOLA --> COBOLB --> COBOLC (which causes a zero divide).

**Figure 66: COBOLB program**
Figure 67 on page 193 generates a divide-by-zero condition. The divide-by-zero condition is percolated back to COBOLB, to COBOLA, and to Language Environment default behavior.

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. COBOLC.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
1   Y  PIC 999.
* PROCEDURE DIVISION USING Y.
   DISPLAY "In COBOLC.".
   COMPUTE Y = 1 / Y.
   GOBACK.
```

Figure 67: COBOLC program

Resuming execution after an IGZ condition occurs

When a COBOL condition with a facility ID of IGZ occurs, you must call CEEMRCR with a 0 or 1 type_of_move before a resume is attempted. You cannot resume in place after an IGZ condition occurs because the current stack frame is that for the runtime library routine. If a user-written condition handler issued a result_code 10 (see “User-written condition handler interface” on page 204) without moving the resume cursor first, that would be a resume in place. A 0 type_of_move results in a resume at the instruction in the program following the call to the COBOL runtime library routine. For example, if you encounter an error when trying to open a file, you cannot resume in place. You must either move the resume cursor and then resume, or percolate the condition.

Resuming execution after a COBOL STOP RUN statement

There is a different constraint on resuming after a COBOL STOP RUN statement. When a STOP RUN is issued, Termination Imminent due to Stop (T_I_S) is raised (see “Processing the T_I_S condition” on page 176 for more information about T_I_S). Therefore, you can respond to a STOP RUN by registering a user-written condition handler to recognize T_I_S.

This condition handler cannot call CEEMRCR with a 0 type_of_move, which means to move the resume cursor to the point in your program just after the STOP RUN statement. This violates the standard definition of a STOP RUN being the last statement to execute in the program in which it is coded. Assuming your program is a subroutine, you could issue a 1 type_of_move to move the resume cursor to the call return point of the stack frame previous to the one of the program that issued the STOP RUN. You could also percolate the condition.

Reentering COBOL programs after stack frame collapse

A stack frame collapse occurs when the condition manager skips over one or more active routines and execution resumes in an earlier routine on the stack. This can occur due to either of the following:

- An explicit GOTO out of block issued from a C or PL/I routine
- Moving the resume cursor using the CEEMCR callable service and requesting a resume

Language Environment resets any intervening COBOL programs from an active to inactive state, provided they are the following:
Condition handling interactions

- VS COBOL II programs compiled with the CMPR2 compiler option
- VS COBOL II programs compiled with NOCMPR2 that do not use nested programs
- COBOL for OS/390 & VM, COBOL for MVS & VM or COBOL/370 programs compiled with the CMPR2 compiler option or
- COBOL for OS/390 & VM, COBOL for MVS & VM and COBOL/370 programs compiled with NOCMPR2 that do not use the combination of the INITIAL attribute, nested programs, and file processing in the same compilation unit
- Enterprise COBOL for z/OS programs that do not use the combination of the INITIAL attribute, nested programs, and file processing in the same compilation unit

After a stack frame collapse, the routines listed above can be reentered. Language Environment issues a warning message during stack frame collapse for each intervening COBOL program that does not adhere to the above restrictions. In addition, after the GOTO or resume is performed, any attempt to re-enter these programs is diagnosed as an attempted recursive entry error.

Handling fixed-point and decimal overflow conditions

The ON SIZE ERROR phrase continues to be invoked by COBOL to handle fixed-point and decimal overflow conditions, regardless of whether these conditions are enabled by Language Environment.

Fortran condition handling semantics

Fortran language syntax provides limited error handling through the ERR and IOSTAT specifiers that can be coded on Fortran I/O statements, and the STAT specifier that can be coded on Fortran ALLOCATE and DEALLOCATE statements. When ERR, IOSTAT, or STAT are present on a statement, and an error is detected, Fortran semantics take precedence over Language Environment condition handling and control returns immediately to the Fortran program.

Language Environment does not support the use of the Fortran global error option table or extended error handling services.

Arithmetic program interruptions from vector instructions

When one of the following arithmetic program interruptions occurs during the execution of a vector instruction, the interaction with a condition handler is equivalent to the corresponding exception for a scalar instruction:

- Fixed-point overflow exception
- Exponent-overflow exception
- Exponent-underflow exception
- Floating-point divide exception
- Unnormalized-operand exception
- Square-root exception

The unnormalized-operand exception occurs only for vector instructions, but the same considerations apply. Exceptions caused by vector instructions or scalar instructions are comparable in terms of the information available to the condition handler and the possible resume and fix-up and resume actions that the condition handler can request.

Whenever a condition handler is entered because a vector instruction caused one of the arithmetic program interruptions, the information available to that handler represents an exception for only a single element involved in the vector instruction. Both the condition token provided directly to the user condition handler and the qualifying data that it can use are the same as for the corresponding scalar instruction exception. None of this information reflects anything about a vector instruction. Therefore, the condition handler must treat the condition as though it were a scalar exception in which the equivalent scalar instruction is simply one of the successive elementary operations that comprise the vector instruction.
The same resume and fix-up and resume actions for scalar conditions can be requested when they apply to one of the operations that comprise the vector instruction. For example, when the **resume with new input value** action (result code 60 with a new condition token of CEE0CE) is allowed for the condition and is requested by the user condition handler to provide a new input value for the failing operation, the new input value is used to reexecute the failing vector instruction. This is identical to providing a new input value for a scalar instruction except that a particular element of a vector register is involved. Similarly, when the **resume with new output value** action (result code 60 with a new condition token of CEE0CF) is allowed for the condition and is requested by the user condition handler to provide a new result for the failing operation, the new result that the user condition handler provides replaces the appropriate element of the vector register. This is identical to providing a new result for a scalar instruction in that the new result replaces whatever the instruction left in its result position; in the vector case, the result position is a particular element of a vector register. For the vector instruction, resumption then occurs by continuing to execute the failing vector instruction but starting with the next element.

Because a vector instruction is semantically equivalent to a loop of elementary operations, more than one arithmetic program interruption can occur for the same vector instruction but for different elements. When this occurs, each exception is presented one at a time as a condition to any condition handlers that are involved.

**Restrictions on using vector instructions in user-written condition handlers**

When a vector instruction causes a program interruption, no vector instructions can be executed from within any user-written condition handler entered for the condition. In addition, if one of these condition handlers incurs another condition, then subordinate user-written condition handlers that are entered for any nested conditions are also prohibited from executing vector instructions. This restriction is not diagnosed and violation of it causes unpredictable results.

**PL/I condition handling semantics**

Enterprise PL/I for z/OS condition handling semantics are the same as PL/I except that Enterprise PL/I for z/OS, like C, ignores any hardware fixed-point overflow exceptions.

When an exception occurs in a PL/I routine, PL/I language semantics for handling the condition prevail. Therefore, the behavior of PL/I condition handling in applications consisting of only PL/I routines is unchanged under Language Environment.

In PL/I, you handle all runtime conditions by writing ON-units. An ON-unit is a procedure that is established in a block when the ON statement for the ON-unit is run. The ON-unit itself runs when the specified condition in the ON statement is raised. The establishment of an ON-unit applies to all dynamically descendent (inherited from calling procedure) blocks of the block that established it; a condition occurring in a called procedure could result in an ON-unit being run in the caller.

This section provides a high-level view of how condition handling works if an exception occurs in a PL/I routine, and only PL/I routines are on the stack. For a more detailed explanation of PL/I condition handling, refer to *PL/I for MVS & VM Language Reference*. For details about how PL/I condition handling works in an ILC application, see *z/OS Language Environment Writing Interlanguage Communication Applications*.

**PL/I condition handling actions**

Refer to Figure 68 on page 196 throughout the following summary of the steps taken to process a condition when there are only PL/I routines on the stack.
1. Assume a condition such as CONVERSION, which is severity 3, occurs in routine 4.

2. Language Environment moves down the stack towards the earliest stack frame. If a PL/I ON-unit is established for the CONVERSION condition, it is given control.

3. If all stack frames have been visited and no ON CONVERSION unit was found, a message is issued. The condition is promoted to the ERROR condition if it meets any of the qualifications listed in “Promoting conditions to the PL/I ERROR condition” on page 196. Otherwise, the PL/I implicit action occurs. A CONVERSION condition would be promoted to ERROR.

4. The Language Environment condition manager makes another pass of the stack, beginning in Routine 4 where the original condition occurred. If a PL/I ERROR ON-unit is established, it is invoked.

5. If either of the following occurs:
   - An ERROR ON-unit is found, but it does not issue a GOTO out of block
   - No ERROR ON-unit is found.
   then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled Condition). T_I_U maps to the PL/I FINISH condition. (See “Termination imminent step” on page 175 for a discussion of T_I_U.)

6. Language Environment makes yet another pass of the stack, beginning in Routine 4 where the original condition occurred. If a PL/I FINISH ON-unit is established, it is invoked.

7. If all stack frames have been visited, and no FINISH ON-unit issued a GOTO out of block, then Language Environment begins thread termination activities in response to the unhandled condition. Since a message was issued for the CONVERSION condition before it was promoted to the ERROR condition, no message is issued at this time.

Promoting conditions to the PL/I ERROR condition

PL/I promotes the following conditions to the PL/I ERROR condition:
- Any PL/I condition for which the implicit action is to promote to the ERROR condition. The appropriate ONCODE is used. See PL/I for MVS & VM Language Reference for details.

Mapping non-PL/I conditions to PL/I conditions

Some non-PL/I conditions map directly to PL/I conditions:
- The Language Environment conditions listed in the first column below map directly to the PL/I conditions in the second column.

  **Attention**
  ATTENTION
  **Decimal divide**
  ZERODIVIDE
  **Decimal overflow**
  FIXEDOVERFLOW

Figure 68: PL/I condition processing
Exponent overflow
OVERFLOW

Exponent underflow
UNDERFLOW

Fixed-point divide
ZERODIVIDE

Fixed-point overflow
FIXEDOVERFLOW

Floating-point divide
ZERODIVIDE

These Language Environment conditions map directly to the PL/I conditions. They are detected by the hardware and are normally represented by condition tokens with a facility ID of CEE when raised. They are represented by an IBM condition token only when signaled by the PL/I SIGNAL statement.

• The following map directly to ERROR:
  – A Language Environment condition of severity 2, 3, or 4 that does not map to one of the PL/I conditions listed above
    For these conditions, an established ERROR ON-unit is run on the first pass of the stack. In general, the ONCODE is 9999. Some Language Environment conditions that map to ERROR, however, are represented by an ONCODE other than 9999. Examples are some of the conditions raised by the Language Environment math services.
  – Any other condition of severity 2, 3, or 4
    For these conditions, an established ERROR ON-unit is run on the first pass of the stack; the ONCODE is 9999.

Additional PL/I condition handling considerations

Keep the following additional PL/I condition handling considerations in mind:

• Non-PL/I conditions of severity 0 or 1 are not promoted to ERROR.
• Promoting any non-PL/I condition to a PL/I condition is prohibited.
• Raising a PL/I condition using the CEESGL callable service is prohibited.
• Issuing a call to CEEMRCR from within a PL/I ON-unit to move the resume cursor is prohibited. But, you can move the resume cursor by using CEEMRCR from within a Language Environment user-written condition handler.

PL/I condition handling example

The following example shows an example of condition handling for PL/I.

```plaintext
/*PROCESS MACRO;
/**Module/File Name: IBMIDIVZ
*******************************************************************************/
/*                                                                   */
/* PL/I Condition Handling Functions:                                */
/*                : Establish ZERODIVIDE ON-unit                     */
/*                : GO TO out of ZERODIVIDE ON-unit                  */
/*                : PL/I Normal return from ZERODIVIDE ON-unit       */
/*                : Revert ZERODIVIDE ON-unit                        */
/*                : PL/I System action on ZERODIVIDE condition       */
/*                                                                   */
/* 1. This example establishes a ZERODIVIDE ON-unit.                 */
/* 2. A subprogram, sdivide, is called and causes a ZERODIVIDE       */
/*    condition to occur.                                            */
/* 3. The ZERODIVIDE ON-unit is entered. A GOTO out of the ON-unit    */
/*    is processed. The program resumes at the label                 */
/*    "after_1st_zerodivide".                                       */
/* 4. A new ZERODIVIDE ON-unit is established and it overrides the    */
/*    current established ZERODIVIDE ON-unit.                       */
/* 5. The subroutine sdivide is called a second time.                */
/* 6. The newly established ZERODIVIDE ON-unit is entered. A GOTO    */
```
**Condition handling interactions**

/* is not executed, and the program resumes at the location */
/* following the instruction that caused the condition. This */
/* is the PL/I normal return action for the ZERODIVIDE condition. */
/* 7. The established ZERODIVIDE ON-unit is canceled by executing */
/* the REVERT ZERODIVIDE statement. */
/* 8. Sdivide is called a third time. Because there is no */
/* ZERODIVIDE ON-unit established, the PL/I implicit action */
/* is executed. Namely, the ERROR condition is raised and the */
/* program is terminated. */
*****************************************************************************/

CEPLCND: Proc Options(Main);

%INCLUDE  CEEIBMAW;
%INCLUDE  CEEIBMCT;
dcl in_zdiv_ou1 char (1), in_zdiv_ou2 char(1),fell_thru char(1);
in_zdiv_ou1 = 'N';
in_zdiv_ou2 = 'N';
fell_thru = 'N';
*****************************************************************************
/* A ZERODIVIDE ON-unit is established when control reaches the */
/* ON statement. */
*****************************************************************************
on zerodivide begin;
in_zdiv_ou1 = 'Y';
go to after_1st_zerodivide;
end;
*****************************************************************************
/* The first call to sdivide will result in the ZERODIVIDE */
/* condition being raised. The preceding established ON-unit */
/* gets control. Due to a GO TO out of the ON-unit, execution */
/* resumes immediately at label after_1st_zerodivide. This is */
/* verified by checking that the flow of control did not resume */
/* at the instruction following the ZERODIVIDE condition. */
*****************************************************************************
call sdivide;after_1st_zerodivide:
if (fell_thru = 'Y') then do;
  put skip list ('Error in flow of control after' || 'the first call to sdivide. ');
end;
*****************************************************************************
/* A new ZERODIVIDE ON-unit is established when control */
/* reaches the following ON ZERODIVIDE statement. */
*****************************************************************************
on zerodivide begin;
in_zdiv_ou2 = 'Y';
end;
*****************************************************************************
/* Subroutine sdivide is called a second time to raise the */
/* ZERODIVIDE condition. Control enters the established */
/* ZERODIVIDE ON-unit. On exit from the preceding zerodivide */
/* ON-unit, control returns to the instruction following the */
/* divide by zero in subroutine SDIVIDE. A check is made to */
/* detect if control flowed to the instruction following the */
/* one that caused the zerodivide condition to be raised. */
*****************************************************************************
call sdivide;
if (fell_thru = 'N') then do;
  put skip list ('Error in flow of control after second call to cepldiv. ');
end;
*****************************************************************************
/* The ZERODIVIDE ON-unit is canceled by action of the */
/* REVERT statement. */
*****************************************************************************
revert zerodivide;
if (in_zdiv_ou1 = 'N' | in_zdiv_ou2 = 'N') then
  put skip list ('Error in flow of control to ON-units');
else do;
  put skip list ('The PL/I condition handling example' || 'will terminate with PL/I message IBM0301');
end;
*****************************************************************************
/* Sdivide is called for the third and final time. Because */
/* there are no established ON-units, the implicit action */
/* for ZERODIVIDE takes place. */
*****************************************************************************
call sdivide;
put skip list ('Error in flow of control after third');
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 15, “Introduction to Language Environment condition handling,” on page 167.

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” on page 199. 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()` (on MVS) 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()` (on MVS) 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.

POSIX signal handling can take effect even if no C routine is present on the stack. For example, a COBOL program calls C routine C1. C1 registers a POSIX signal catcher, then C1 returns control to the COBOL program. The registered POSIX signal handler would still be present to handle a POSIX signal even though the stack no longer contains a C stack frame.

**Synchronous POSIX signal and Language Environment condition handling interactions**

This topic discusses how Language Environment processes most synchronous POSIX signals. (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 202, 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:

- A signal you generate by calling the CEESGL (signal a condition) callable service
- 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.

- 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 imminent, as described in “Enablement step for signals under z/OS UNIX” on page 200, “Condition step for POSIX signals under Language Environment” on page 201, and “Termination imminent step under z/OS UNIX” on page 202.

Enablement step for signals under z/OS UNIX

Figure 69 on page 201 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 202 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 170). From there, Language Environment either disables the signal, or passes it into the Language Environment condition step.
Condition step for POSIX signals under Language Environment

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

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 a user-written condition handler registered using the CEEHDLR callable service is present on the stack frame, Language Environment gives it a chance to handle the condition.
   - 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 34 on
     page 174). If the condition that the signal represents is of severity 2 or greater, Language
     Environment promotes the condition to Termination Imminent due to an Unhandled Condition
     (T_I_U).

Termination imminent step under z/OS UNIX

In a POSIX(ON) environment, Language Environment’s termination imminent step takes place as
described in “Termination imminent step” on page 175, 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 CEESGL, 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.

IBM extensions to POSIX signals that do not go through condition handing:
   • SIGDUMP cannot be caught or ignored; it always takes effect.
   • SIGTHSTOP and SIGTHCONT cannot be caught or ignored; they always take effect.
Chapter 17. Coding a user-written condition handler

This topic describes how you can code a user-written condition handling routine and provides examples for Language Environment-conforming HLLs.

Your user-written condition handler can test for the occurrence of a particular condition by coding a 12-byte condition token or by coding a symbolic feedback code. You can use the Language Environment callable service CEEHDLR to register the condition handler. For information about using CEEHDLR, see “User-written condition handler interface” on page 204.

The USRHDLR runtime option enables you to register a user-written condition handler at stack frame 0 without having to recompile your application to include a call to CEEHDLR. This is particularly useful in supporting Fortran applications because Fortran applications are unable to directly call CEEHDLR.

Nested conditions can be used in your routine as long as the language your routine is written in allows it to be recursively entered. You should design the routine to handle specific conditions rather than designing the routine to handle a wide variety of conditions. You should also code the condition handling routine to respond to the original condition on the first pass of the stack, rather than coding a routine to handle T_I_U on the second pass of the stack. This helps ensure that the handling that you perform addresses the original condition. The more specific the condition is that you design the handler for, the more precise the fix can be.

PL/I considerations

User condition handlers can now be written in PL/I; that is, you can register a PL/I external procedure as a user-written condition handler using the Language Environment callable service CEEHDLR, and unregister it using CEEHDLU.

Restrictions on PL/I user-written condition handlers are:

• If a user handler is registered in the PL/I main routine, it must be unregistered using CEEHDLU before the main returns via a RETURN statement or by reaching the END statement. One implication is that a user handler registered in the main routine does not gain control for the PL/I FINISH condition raised due to normal termination of the main routine.

• You cannot collapse multiple BEGIN blocks using a RETURN statement when CEEHDLR has been invoked within a nested block.

• The following condition handling pseudovariables and built-in functions are still restricted to PL/I ON-units and are not available in user handlers:
  – DATAFIELD
  – ONCHAR
  – ONCODE
  – ONCOUNT
  – ONFILE
  – ONKEY
  – ONLOC
  – ONSOURCE

• User-written condition handlers are not supported in PL/I multitasking applications.

Invocation of a procedure registered as a user handler

A PL/I parameter declared as a structure expects an extra PL/I descriptor; however, Language Environment passes argument lists by reference and has no knowledge of PL/I descriptors. Therefore for
the parameter list to be received, declare the parameters with the OPTIONS(BYVALUE) option as shown in Figure 70 on page 204.

```
PLIHDLR: PROC(ptr1, ptr2, ptr3, ptr4) OPTIONS(BYVALUE);
  DCL (ptr1, ptr2, ptr3, ptr4)  POINTER;
  DCL 1 Current_condition       BASED(ptr1)
    ;
  DCL Token FIXED BIN(31)       BASED(ptr2);
  DCL Result_code FIXED BIN(31) BASED(ptr3);
  DCL 1 New_condition           BASED(ptr4)
    ;
```

*Figure 70: Parameter declarations in a PL/I user-written condition handler*

**Types of conditions you can handle**

A user-written condition handler can, in general, intercept and process any condition, regardless of the language of the routine in which the condition occurred. This means that you can code a user-written condition handler to respond to condition tokens with any of the following facility IDs:

- CEE, representing Language Environment and POSIX-defined conditions
- EDC, representing C and C++ conditions
- IGZ, representing COBOL conditions
- FOR, representing Fortran conditions
- IBM, representing PL/I conditions

In general, your user-written condition handler can use any of the Language Environment condition handling services. Specific exceptions follow:

- The ways in which you can resume after an IGZ condition of severity 2 or above are restricted. See “Resuming execution after an IGZ condition occurs” on page 193 for details.

- If an IBM condition of severity 2 or above was raised, then you cannot issue a resume without first moving the resume cursor.

  This restriction does not apply to IBM conditions of severity 0 or 1, or any IBM conditions signaled using the PL/I SIGNAL statement.

- You cannot promote any condition to an IBM condition (one that belongs to PL/I). You can promote IBM conditions to conditions with facility IDs of CEE, EDC, FOR, or IGZ.

For more information about coding user-written condition handlers to respond to conditions of different facility IDs, see “Using symbolic feedback codes” on page 236.

**User-written condition handler interface**

Use CEEHDLR to register a user-written condition handler. See *z/OS Language Environment Programming Reference* for more information about CEEHDLR.

User-written condition handlers are automatically unregistered when the stack frame they're associated with is removed from the stack due to a return, GOTO out of block, or a move of the resume cursor. You can, however, call CEEHDLU to explicitly unregister a user-written condition handler. See *z/OS Language Environment Programming Reference* for more information about CEEHDLU.

Recursion is allowed if a handler is registered within a handler, and nested conditions are allowed.
It is invalid to promote a condition without returning a new condition token. You cannot promote a condition to a PL/I condition.

**c_c tok (input)**
A 12-byte condition token that identifies the current condition being processed. Language Environment uses this parameter to tell your condition handler what condition has occurred.

**token (input)**
A 4-byte integer that specifies the token you passed into Language Environment when this condition handler was registered by a call to the CEEHDLR callable service.

**result_code (output)**
A 4-byte integer that contains instructions about responses the user-written condition handler wants Language Environment to make when processing the condition. The `result_code` is passed by reference. Valid responses are shown in Table 40 on page 205.

<table>
<thead>
<tr>
<th>Response</th>
<th>Result_Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>resume</td>
<td>10</td>
<td>Resume at the resume cursor (condition has been handled). Unless the resume cursor has been moved, this response can only be used if the condition being handled specifically allows this form of resumption.</td>
</tr>
<tr>
<td>percolate</td>
<td>20</td>
<td>Percolate to the next condition handler. If a <code>result_code</code> is not explicitly set by a handler, this is the default <code>result_code</code>.</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>Percolate to the first user-written condition handler for the stack frame that is before the one to which the handle cursor points. This can skip a language-specific condition handler for this stack frame as well as any remaining user-written condition handlers in the queue for this stack frame.</td>
</tr>
<tr>
<td>promote</td>
<td>30</td>
<td>Promote to the next condition handler.</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>Promote to the stack frame before the one to which the handle cursor points. This can skip a language-specific condition handler for this stack frame as well as any remaining user-written condition handler in the queue at this stack frame.</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>Promote and restart condition handling at the first condition handler of the stack frame of the handle cursor.</td>
</tr>
<tr>
<td>fix-up and resume</td>
<td>60</td>
<td>Provide the fix-up actions indicated by <code>new_condition</code> and by any qualifying data values that apply to the condition; then resume execution. This response is only allowed if the resume cursor has not been moved and only if the condition being handled allows this response. <code>new_condition</code> must be set by the condition handler to request one of the specific actions for the condition.</td>
</tr>
</tbody>
</table>

If `result_code` is not explicitly set by the handler, the default response is Value=20, Percolate to the next condition handler.
new_condition (output)
A 12-byte condition token that represents either the promoted condition for a promote response (result_code values of 30, 31, and 32) or the requested fix-up actions for a fix-up and resume response (result_code value of 60).

When a result_code of 60, denoting fix-up and resume, is set by the condition handler, new_condition must be set to a condition token that indicates what fixup action is requested. Many conditions, including mathematical routines, use the condition tokens in Table 41 on page 206 to resume with corrective action (either resume with new input value or resume with new output value). For some conditions, there may be other condition tokens that can be provided by the condition handler in new_condition to request specific fixup actions.

<table>
<thead>
<tr>
<th>Symbolic feedback code (fc)</th>
<th>Severity</th>
<th>Message number</th>
<th>Fixup action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE0CE</td>
<td>1</td>
<td>398</td>
<td>Fixup with new input value. The service that signaled the condition is invoked again with the new argument value provided by the handler as qualifying data.</td>
</tr>
<tr>
<td>CEE0CF</td>
<td>1</td>
<td>399</td>
<td>Fixup with new output value. The service that signaled the condition returns as its result, the value provided by the handler as qualifying data.</td>
</tr>
</tbody>
</table>

Registering user-written condition handlers using USRHDLR
Use the USRHDLR runtime option to register a user-written condition handler to run at one of the following times (or both):

- At stack frame 0 – the condition handler specified as lmname (see syntax below) is invoked after the default HLL condition handler for the main program, but before the HLL condition handler for stack frame 0. The condition percolated or promoted by this user-written condition handler is not passed to any other condition handler.
- The condition handler specified as lmname2 is given control after each condition completes the enablement phase, but before any other registered user condition handler is given control.

NOUsrhdlr
Specifies that no user-written condition handler is registered.

USrhdlr
Specifies that a user-written condition handler is registered.

lmname
The entry point name or alias name of a load module that contains the user-written condition handler to be registered at stack frame 0.

lmname2
The entry point name or alias name of a load module that contains the user-written condition handler to be registered to get control after the enablement phase and before any other condition handler.
The condition handlers registered by the USRHDLR runtime option can return any of the result codes allowed for a condition handler registered with the CEEHDLR callable service.

For more information about using the USRHDLR runtime option, see z/OS Language Environment Programming Reference.

**Nested conditions**

A nested condition is one that occurs within a C/C++ signal handler, PL/I ON-unit, or user-written condition handler invoked to handle a condition. When conditions occur during the condition handling process, the handling of the original condition is suspended and further action is taken based on the state of the condition handling.

The DEPTHCONDLMT runtime option indicates whether nested conditions are permitted while your application runs. If you specify DEPTHCONDLMT(1), handling of the initial condition is allowed, but any additional nested condition causes your application to abend. If you specify DEPTHCONDLMT(0), an unlimited number of nested conditions is permitted. If you specify some other integer value for DEPTHCONDLMT, Language Environment allows handling of the initial condition plus additional levels of nested conditions before your application abends (see z/OS Language Environment Programming Reference for more information).

If a nested condition is allowed within a user-written condition handler, Language Environment begins handling the most recently raised condition. After the most recently raised condition is properly handled, execution begins at the instruction pointed to by the resume cursor, the instruction following the point where the condition occurred. If a user-written condition handler is registered using CEEHDLR within another user condition handler, nested conditions are handled by the most recently registered condition handler.

If any HLL or user-written condition handler moves the resume cursor closer to the oldest stack frame both conditions are considered handled. The application resumes running at the instruction pointed to by the resume cursor. The resume cursor can be moved using the CEEMRCR callable service, or by language constructs such as GOTO.

**Nested conditions in applications containing a COBOL program**

You must take special care when dealing with nested conditions in ILC applications. For example, the following scenario can cause your application to abend:

1. A nested condition occurs within a COBOL user-written condition handler (CBLUHDLR). Condition handlers written in COBOL must be compiled with Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370.

2. The COBOL user-written condition handler calls another user-written condition handler established using CEEHDLR to handle the nested condition.

3. The user-written condition handler percolates the condition.

In this scenario, the condition can be percolated back to the stack frame where the original condition occurred. Since condition handling actions for the routine where the condition originally occurred include calling CBLUHDLR, CBLUHDLR can be recursively entered. This is not permitted under COBOL/370, and your application abends.

If CBLUHDLR is compiled with Enterprise COBOL for z/OS, COBOL for OS/390 & VM, or COBOL for MVS & VM; then the recursive call is allowed if RECURSIVE is specified in the PROGRAM-ID. A rule of thumb is to ensure that COBOL user-written condition handlers that call other user-written condition handlers do not regain control or, make sure they are capable of being recursively entered.

**Using Language Environment condition handling with nested COBOL programs**

If your application contains both nested COBOL programs and calls to Language Environment condition handling services, keep the following restrictions in mind:
• Do not call CEEHDLR from a nested COBOL program.
• Do not call CEEMRCR with a 1 type_of_move from a user handler associated with a stack frame that was called by a nested COBOL program. In Figure 71 on page 208, Program A calls nested Program B. Program B calls Program C, which registers a user-written condition handler, UWCHC. UWCHC cannot call CEEMRCR with a 1 type_of_move, which would move the resume cursor back to nested Program B.

![Diagram of program flow](image)

Figure 71: Restricted type_of_move If COBOL nested programs are present

### Examples with a registered user-written condition handler

This section contains C, C++, COBOL, PL/I, and assembler examples in which user-written condition handlers are registered to respond to specific conditions that might occur in an application.

- In “Signaling and handling a condition in a C/C++ routine” on page 223, C or C++ call CEEHDLR CEEGQDT, and CEEMRCR to respond to a signaled condition.
- In “Handling a divide-by-zero condition in a COBOL program” on page 225, COBOL calls CEEHDLR, CEE3GRN, and CEEMOUT to respond to the significance condition (which was enabled using CEE3SPM).
- In “Handling a program check in an assembler routine” on page 229, assembler calls CEEHDLR to register a condition handler that responds to a program check.

### Handling a divide-by-zero condition in C, C++, COBOL, or PL/I

Figure 72 on page 209 and the following examples provide an illustration of how user-written condition handlers can handle conditions such as a divide-by-zero in a C, C++, COBOL, or PL/I application. In the C or C++ examples in “C or C++ handling a divide-by-zero condition” on page 209, the COBOL examples in “COBOL handling a divide-by-zero condition” on page 211, and the PL/I examples in “PL/I handling a divide-by-zero condition” on page 213, the main routine calls CEEHDLR to register the user-written condition handler (“USRHDLR program (COBOL)” on page 212 for COBOL). The main routine then calls the DIVZERO routine (Figure 73 on page 212 for COBOL), in which a divide-by-zero exception occurs.
Divide-by-zero is enabled as a condition in the following steps:

1. The handle cursor, which first points at DIVZERO’s stack frame, moves down the stack to the USRHDLR condition handler, the first user-written condition handler established to handle conditions for the main routine’s stack frame.

2. For divide-by-zero conditions, USRHDLR issues a call to CEEMRCR (Move Resume Cursor Relative to Handle Cursor) with a 0 type_of_move, meaning move the resume cursor to the call return point of the stack frame associated with the handle cursor. (The call return point is the next instruction after the call to the DIVZERO routine.)

3. Execution resumes in the main routine at this point. A divide-by-zero condition is the only type of program interrupt for which USRHDLR causes a resume.

4. All other program interrupts are percolated to the next condition handler on the stack.

For simplicity, the examples shown in this topic do not include calls to some Language Environment services that could also be useful for handling conditions in your application. For example, you might code in the USRHDLR routine a call to the CEE3GRN callable service in order to get the name of the routine that incurred the condition.

**C or C++ handling a divide-by-zero condition**

The following example contains the C/C++ routine that performs the tasks involved with handling a divide-by-zero condition in C, C++, COBOL, or PL/I.

```c
#pragma noinline(divzero)
/*Module/File Name:  EDCDIVZ */
/**********************************************************/
/*                                                        */
/*  MAIN                 .-> DIVZERO                      */
/*  - register handler   |   - force a divide-by-zero     */
/*  - call DIVZERO     --'                                */
/*  ==> "resume point"                                    */
/*  - unregister handler                                  */
/*                         USRHDLR:                       */
/*                         - if divide-by-zero            */
/*                           - move resume cursor         */
/*                           - resume at "resume point"   */
/*                                                        */
/**********************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>
#ifdef __cplusplus
extern "C" {
#endif
void usrhdlr(_FEEDBACK *, _INT4 *, _INT4 *, _FEEDBACK *);
```

**Figure 72: Handle and resume cursor movement as a condition is handled**
```c
#ifndef __cplusplus
}
#endif

void divzero(int);

int main(void) {
  _FEEDBACK fc;
  _INT4 divisor;
  _INT4 token;
  _ENTRY pgmptr;
  /* Register a user-written condition handler. */
  pgmptr.address = (_POINTER)&usrhdlr;
  pgmptr.nesting = NULL;
  token = 97; CEEHDLR (&pgmptr, &token, &fc);
  if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
    printf( "CEEHDLR failed with message number %d\n",
            fc.tok_msgno);
    exit(99);
  }
  printf("MAIN: Registered USRHDLR.\n");
  /* Call DIVZERO to divide by zero and drive USRHDLR */
  divisor = 0;
  divzero(divisor);
  printf("MAIN: Resumption after DIVZERO.\n");
  /* Unregister the user condition handler. */
  CEEHDLU (&pgmptr, &fc);
  if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
    printf( "CEEHDLU failed with message number %d\n",
            fc.tok_msgno);
    exit(99);
  }
  printf("MAIN: Unregistered USRHDLR.\n");
  } /* end main */
  void divzero(int arg) {
    printf("  DIVZERO: Starting.\n");
    arg = 1 / arg;
    printf("  DIVZERO: Returning to its caller.\n");
  } /* end divzero */

  /**************************************************************************/
  /* usrhdlr will handle DIVIDE-BY-ZERO conditions... */
  /* all others will be percolated. */
  /**************************************************************************/
  void usrhdlr(_FEEDBACK *cond,_INT4 *input_token,
               _INT4 *result, _FEEDBACK *new_cond) {
    _INT4 move_type_0 = 0;
    _INT4 move_type_1 = 1;
    _FEEDBACK feedback;
    /* values for handling the conditions */
    #define resume 10
    #define percolate 20
    #define promote 30
    #define promote_sf 31
    printf(">>> USRHDLR: Entered User Handler \n");
    printf(">>> passed token value is %d\n",*input_token);
    /* check if the DIVIDE-BY-ZERO message (0C9) */
    if (cond->tok_msgno == 3209) {
      CEEMRCR (&move_type_0, &feedback);
      if ( _FBCHECK ( feedback , CEE000 ) != 0 ) {
        printf( "CEEMRCR failed with message number %d\n",
                feedback.tok_msgno);
        exit(99);
      }
      *result = resume;
      printf(">>> USRHDLR: Resuming execution\n");
    } else {
      /* not DIVIDE-BY-ZERO */
      *result = percolate;
      printf(">>> USRHDLR: Percolating it\n");
    }
  } /* end usrhdlr */
```
COBOL handling a divide-by-zero condition

The program in the following example registers a user-written condition handler, calls the DIVZERO subroutine, and unregisters the condition handler on return from the subroutine.

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. EXCOND.
DATA DIVISION.
WORKING-STORAGE SECTION.
  77 DIVISOR PIC S9(9) BINARY.
** Declarations for condition handling
** 77 TOKEN PIC X(4).
  77 PGMPTR USAGE IS PROCEDURE-POINTER.
  01 FC.
    02 Condition-Token-Value.
      COPY CEEIGZCT.
      03 Case-1-Condition-ID.
        04 Severity PIC S9(4) BINARY.
        04 Msg-No PIC S9(4) BINARY.
        03 Case-2-Condition-ID
          REDEFINES Case-1-Condition-ID.
          04 Class-Code PIC S9(4) BINARY.
          04 Cause-Code PIC S9(4) BINARY.
        03 Case-Sev-Ctl PIC X.
        03 Facility-ID PIC XXX.
      02 I-S-Info PIC S9(9) BINARY.
PROCEDURE DIVISION.
PARA-CND01A.
** Register a user-written condition handler.**
******************************************************************************
SET PGMPTR TO ENTRY "USRHDLR".
MOVE ZERO TO TOKEN.
CALL "CEEHDLR" USING PGMPTR TOKEN FC.
  IF CEE000 OF FC THEN
    DISPLAY "EXCOND: REGISTERED USRHDLR."
  ELSE
    DISPLAY "CEEHDLR failed with msg "
      Msg-No OF FC UPON CONSOLE
    STOP RUN
  END-IF.
******************************************************************************
** Call DIVZERO to force a divide-by-zero and drive USRHDLR **
****************************************************************************
MOVE 00 TO DIVISOR.
CALL "DIVZERO" USING DIVISOR.
DISPLAY "EXCOND: RESUMED AFTER DIVZERO."
****************************************************************************
** Unregister the user-written condition handler.**
****************************************************************************
CALL "CEEHDLU" USING PGMPTR FC.
  IF CEE000 OF FC THEN
    DISPLAY "EXCOND: UNREGISTERED USRHDLR."
  ELSE
    DISPLAY "CEEHDLU failed with msg "
      Msg-No OF FC UPON CONSOLE
    STOP RUN
  END-IF.
GOBACK.
END PROGRAM EXCOND.
```
Figure 73 on page 212 shows the subroutine DIVZERO that generates the divide-by-zero condition.

```
CBL LIB,QUOTE,NODYNAM,NOOPT
*Module/File Name: IGZTDIVS
IDENTIFICATION DIVISION.
PROGRAM-ID. DIVZERO.

DATA DIVISION.
LINKAGE SECTION.
01 ARG PIC S9(9) BINARY.

PROCEDURE DIVISION USING ARG.
   DISPLAY " DIVZERO: STARTING.".
   COMPUTE ARG = 1 / ARG.
   DISPLAY " DIVZERO: RETURNING TO ITS CALLER.".
   GOBACK.
END PROGRAM DIVZERO.
```

**USRHDLR program (COBOL)**

The following example shows the user-written condition handler registered by EXCOND to handle the divide-by-zero condition. When the divide-by-zero condition arises, USRHDLR calls CEEMRCCR with a 0 type of move. Doing so moves the resume cursor to the point in EXCOND after the call to DIVZERO.

```
CBL LIB,QUOTE
*Module/File Name: IGZTDIVU
****************************************************
*                                                    *
* USRHDLR                                            *
*                                                    *
****************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. USRHDLR.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MISC-VARIABLES.
   02 MOVE-TYPE-0 PIC S9(9) BINARY VALUE ZERO.
   02 MOVE-TYPE-1 PIC S9(9) BINARY VALUE 1.
01 FEEDBACK.
   02 FB-SEVERITY PIC 9(4) BINARY.
   02 FB-DETAIL PIC X(10).

*     LINKAGE SECTION.
****************************************************
   *     *
   * Note: the symbolic names of the condition tokens *
   * for S/370 program interrupt codes 0C1 thru 0CF  *
   * are CEE341 through CEE34F                        *
   *                                                *
****************************************************
01 TOKEN PIC X(4).
01 RESULT-CODE PIC S9(9) BINARY.
   88 RESUME VALUE +10.
   88 PERCOLATE VALUE +20.
   88 PERC-SF VALUE +21.
   88 PROMOTE VALUE +30.
   88 PROMOTE-SF VALUE +31.
01 CURRENT-CONDITION.
   02 Condition-Token-Value.
      COPY CEEIGZCT.
         03 Case-1-Condition-ID.
             04 Severity PIC S9(4) BINARY.
             04 Msg-No PIC S9(4) BINARY.
         03 Case-2-Condition-ID.
            REDEFINES Case-1-Condition-ID.
               04 Class-Code PIC S9(4) BINARY.
               04 Cause-Code PIC S9(4) BINARY.
         03 Case-Sev-Ctl PIC X.
         03 Facility-ID PIC XXX.
```
PL/I handling a divide-by-zero condition

The following example shows the PL/I program that performs the tasks in “Handling a divide-by-zero condition in C, C++, COBOL, or PL/I” on page 208.
The following example is the usrhdlr program (PL/I) to handle divide-by-zero conditions.

```pli
/* Register a user-written condition handler */
token = 97;
Call ceehdlr(Usrhdlr, token, fback);
If fbcheck (fback, cee000) then
    display ('MAIN: registered USRHDLR');
else
    do;
        display ('CEEHDLR failed with message number ' ||
            fback.MsgNo);
        stop;
    end;
/* Call DIVZERO to divide by zero */
divisor = 0;
call divzero (divisor);
display ('MAIN: resumption after DIVZERO');
/* Unregister the user condition handler */
Call ceehdlu (Usrhdlr, fback);
If fbcheck (fback, cee000) then
    display ('MAIN: unregistered USRHDLR');
else
    do;
        display ('CEEHDLU failed with message number ' ||
            fback.MsgNo);
        stop;
    end;
/* Subroutine that simply raises ZERODIVIDE */
divzero: proc (arg);
    dcl arg fixed bin(31);
    display('  DIVZERO: starting.);
    arg = 1 / arg;
    display('  DIVZERO: Returning to its caller');
end divzero;
end Excond;
```

The following example is the usrhdlr program (PL/I) to handle divide-by-zero conditions.
Handling an out-of-storage condition in C, C++, COBOL, or PL/I

You can use the Language Environment condition handling services to resolve an out-of-storage condition in your application. In the user-written condition handler examples that follow, CEEGTST and CEECZST are used to get and reallocate heap storage. CEEMRCR is also used to handle an out-of-storage condition in a user subroutine, and allow the subroutine to be invoked again. For the user code that corresponds to this scenario, see:

- The examples in “C/C++ examples using CEEHDLR, CEEGTST, CEECZST, and CEEMRCR” on page 216 for C or C++
- The examples in “COBOL examples using CEEHDLR, CEEGTST, CEECZST, and CEEMRCR” on page 218 for COBOL
- The examples in “PL/I examples using CEEHDLR, CEEGTST, CEECZST, and CEEMRCR” on page 221 for PL/I

in which:

1. The out-of-storage condition arises in your subroutine, and Language Environment gives control to the user-written condition handler you have registered through CEEHDLR for the out-of-storage condition.
2. The condition handler detects the out-of-storage condition and calls CEEMRCR to set the resume cursor to resume execution at the return address of your subroutine call.
3. On return from the user condition handler, your main program regains control as if your subroutine has actually run.
4. The main program tests a completion indicator and discovers that the subroutine did not actually complete.
5. Your program then recognizes that it has been invoked with insufficient storage for maximum efficiency, and frees some previously allocated storage.
6. The subroutine is invoked a second time and completes successfully.

See z/OS Language Environment Programming Reference for the syntax of all Language Environment condition handling services.
C/C++ examples using CEEHDLR, CEEGTST, CEECZST, and CEEMRCR

The following routine calls CEEHDLR to register a user-written condition handler for the out-of-storage condition, calls CEEGTST to allocate heap storage, and calls CEECZST to alter the size of the heap storage requested.

```c
/**Module/File Name: EDCOOSR */
/*****************************************************************************/
/* M o d u l e  F i l e  N a m e  : E D C O O S R  */
/*****************************************************************************/
/*                                     */
/* Function  : CEEHDLR - Register user condition handler */
/*           : CEEGTST - Get Heap Storage */
/*           : CEECZST - Change the size of heap element */
/*                                     */
/* 1. A user condition handler CECNDHD is registered. */
/* 2. A large amount of HEAP storage is allocated. */
/* 3. A function sub() is called that is known to */
/*    require a large amount of storage. It is not */
/*    known whether the storage for sub() is */
/*    available during this run of the application. */
/* 4. If sufficient storage for sub() is not available, */
/*    a storage condition is generated by Language */
/*    Environment. */
/* 5. CECNDHD gets control and sets resume at the */
/*    next instruction following the call to sub(). */
/* 6. A test for completion of sub() is made after */
/*    the function call. If sub() did not complete, a */
/*    large amount of storage is freed, and sub() is */
/*    invoked a second time. */
/* 7. sub() runs successfully once it has enough storage */
/*    available. */
/*                                     */
/* Note: In order for this example to complete */
/* successfully, the FREE suboption of the HEAP */
/* runtime option must be in effect. */
/*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>
#define BIGSTOR 300000
#define BIGINDX BIGSTOR-1
#ifdef __cplusplus
extern "C" {
#endif

void CECNDHD(_FEEDBACK *, _INT4 *, _INT4 *, _FEEDBACK *);

#ifdef __cplusplus}
#endif

char *sub( );

void main ()
{
    _FEEDBACK feedback;
    _ENTRY pgmptr;
    _POINTER addrss;
    _INT4 token;
    _INT4 hpsize;
    _INT4 heapid;
    _INT4 newsize;
    char *RAN;
    /*******************************************************************/
    /* Call CEEHDLR to register user condition handler CECNDHD.*/
    /*******************************************************************/
    pgmptr.address = (_POINTER)&CECNDHD;
    pgmptr.nesting = NULL;
    token = 97;
    CEEHDLR(&pgmptr, &token, &feedback);
    if ( _FBCHECK ( feedback , CEE000 ) != 0 )
        printf( "CEEHDLR failed with message number %d\n", feedback.tok_msgno);
    else
        printf( "Condition handler registered\n" );
    /*******************************************************************/
    /* Call function sub(). When it becomes active, an out- */
    /* of-storage condition arises if the region is too small. */
    /*******************************************************************/
    return(0);
}
#endif
```
heapid = 0;
hpsize = BIGSTOR;
CEEGTST ( &heapid , &hpsize , &addrss , &feedback );
if ( _FBCHECK ( feedback , CEE000 ) != 0 )
    printf("CEEGTST failed with message number %d\n", feedback.tok_msgno);
RAN = sub ( );
if (RAN != "r")
{
    /**************************************************************************/
    /* If sub() did not run, reduce the size of allocated */
    /* storage and call it a second time. */
    /**************************************************************************/
    newsize = 2000;
    CEECZST ( &addrss, &newsize, &feedback );
    if ( _FBCHECK ( feedback , CEE000 ) != 0 )
        printf("CEECZST failed with message number %d\n", feedback.tok_msgno);
    printf("Function sub is called for the 2nd time\n");
    RAN = sub ( );
    printf("Function sub ran successfully\n", *RAN);
}
} /* end of main */

When any condition occurs in the main routine, user condition handler CECNDHD in the following routine
receives control and tests for the out-of-storage condition. If the out-of-storage condition has occurred,
then CECNDHD calls CEEMRCR to return to the instruction in the main routine after the call to function
sub() that produced the out-of-storage condition.

#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>
#define RESUME 10
#define PERCOLATE 20
#define PROMOTE 30
#define PROMOTE_STACK_FRAME 31
#ifdef _cplusplus
extern "C"
#endif
void CECNDHD (_FEEDBACK *, _INT4 *, _INT4 *, _FEEDBACK *);
#endif
#endif

void CECNDHD (_FEEDBACK *cond, _INT4 *input_token,
    _INT4 *result, _FEEDBACK *new_cond)
{
    _FEEDBACK feedback;
    _INT4 movetyp;
    /**************************************************************************/
    /* Determine if entry was for OUT-OF-STORAGE condition. */
    /**************************************************************************/
    if ( _FBCHECK (*cond , CEE0PD) == 0 )
User-written condition handler

```c
{ printf("SUB not run because of storage condition.\n"); /* Call CEEMRCC to move resume cursor. */ /**************************************************************************/ movetyp = 0; CEEMRCC ( &movetyp, &feedback ); if ( _FBCHECK ( feedback, CEE000 ) != 0 ) { *result = PERCOLATE; } else { *result = RESUME; } else /* Percolate all conditions except for OUT-OF-STORAGE. */ /**************************************************************************/ { *result = PERCOLATE; }
}
```

**COBOL examples using CEEHDLR, CEEGTST, CEECZST, and CEEMRCC**

The following program calls CEEHDLR to register a user-written condition handler for the out-of-storage condition, calls CEEGTST to allocate heap storage, and calls CEECZST to alter the size of the heap storage requested.

```
CBL LIB,QUOTE,NODYNAM
**Module/File Name: IGZTOOSR
******************************************************************************
**  CECNDXP - Call the following Language Environment services:
**  CEEHDLR - Register user condition handler
**  CEEGTST - Get Heap Storage
**  CEECZST - Change the size of heap element

**  1. A user condition handler CECNDHD is registered.
**  2. A large amount of HEAP storage is allocated.
**  3. A subroutine CESUBXP is called that is known to require a large amount of storage. It is not known whether the storage for CESUBXP is available during this run of the application.
**  4. If sufficient storage for CESUBXP is not available, a storage condition is generated by Language Environment.
**  5. CECNDHD gets control and sets resume at the next instruction following the call to CESUBXP.
**  6. A test for completion of CESUBXP is made after the subroutine call. If CESUBXP did not complete, a large amount of storage is freed, and CESUBXP is invoked a second time.
**  7. CESUBXP runs successfully once it has enough storage available.

**  Note: In order for this example to complete successfully, the FREE suboption of the HEAP runtime option must be in effect.******************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. CECNDXP.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 TOKEN PIC X(4).
  01 HEAPID PIC S9(9) BINARY.
  01 HPSIZE PIC S9(9) BINARY.
  01 NEWSIZE PIC S9(9) BINARY.
  01 ADDRSZ PIC S9(9) BINARY.
  01 PGMPTR USAGE IS PROCEDURE-POINTER.
  01 FEEDBACK.
    02 Condition-Token-Value.
    COPY CEEIGZCT.
    03 Case-1-Condition-ID.
      04 Severity PIC S9(4) BINARY.

```

218 z/OS: Language Environment Programming Guide
04  Msg-No      PIC S9(4) BINARY.
  03  Case-2-Condition-ID
       REDEFINES Case-1-Condition-ID.
  04  Class-Code  PIC S9(4) BINARY.
  04  Cause-Code  PIC S9(4) BINARY.
  03  Case-Seq-Ctl  PIC X.
  03  Facility-ID  PIC XXX.
  02  I-S-Info            PIC S9(9) BINARY.
  01  COMPLETED               PIC X.
  88  RAN                 VALUE "Y".
  88  NOTRUN              VALUE "N".
PROCEDURE DIVISION.
0001-BEGIN-PROCESSING.
************************************************************
**  Register user condition handler CECNDHD using CEEHDLR.  **
************************************************************
** Register user condition handler CECNDHD using CEEHDLR.  **
SET PGMPTR TO ENTRY  "CECNDHD".
MOVE 97 TO TOKEN
CALL "CEEHDLR" USING PGMPTR TOKEN.
MOVE 0 TO HEAPID.
************************************************************
** Allocate large amount of heap storage.  **
************************************************************
MOVE  500000 TO HPSIZE.
CALL "CEEGTST" USING HEAPID, HPSIZE, ADDRSS, FEEDBACK.
IF CEE000 OF FEEDBACK THEN
************************************************************
**       Call CESUBXP, which requires a large stack. **
************************************************************
SET NOTRUN TO TRUE
CALL "CESUBXP" USING COMPLETED
************************************************************
**        Check whether CESUBXP completed, or failed with  **
**        storage condition.  If CESUBXP did not run,      **
**        resize the heap element down by a large amount  **
**        and call it again.                             **
************************************************************
IF NOTRUN THEN
   DISPLAY "Reduce storage acquired BY main program"
   " AND CALL CESUBXP again."
   MOVE 300 TO NEWSIZE
   CALL "CEECZST" USING ADDRSS, NEWSIZE
   CALL "CESUBXP" USING COMPLETED
END-IF
ELSE
   DISPLAY "Call TO GET Storage Failed WITH MESSAGE "
   Msg-No OF FEEDBACK
END-IF.
GOBACK.
END PROGRAM CECNDXP.

When any condition occurs in the main program, the user condition handler CECNDHD (see the following
program) receives control and tests for the out-of-storage condition. If the out-of-storage condition has
occurred, then CECNDHD calls CEEMRCR to return to the instruction in the main program after the
subroutine call that produced the out-of-storage condition.

CBL LIB,QUOTE,NODYNAM
*Module/File Name: IGZTOOSH
******************************************************************************
* CECNDHD - Call CEEMRCR to move the resume cursor relative to the handle cursor. *
* CECNDHD is a user condition handler that is registered by the program CECNDXP. CECNDHD gets control from the *
* condition manager and tests for the STORAGE CONDITION. *
* If a STORAGE CONDITION is detected, the resume cursor is moved so that control is returned to the caller of *
* the routine encountering the STORAGE CONDITION. *
******************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. CECNDHD.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Movetype       PIC S9(9) BINARY.
01  Feedback.
02  Condition-Token-Value.
User-written condition handler

The following program is a COBOL subroutine that causes the out-of-storage condition.

```
COPY CEEIGZCT.
03 Case-1-Condition-ID.
   04 Severity   PIC S9(4) BINARY.
   04 Msg-No     PIC S9(4) BINARY.
03 Case-2-Condition-ID
   REDEFINES Case-1-Condition-ID.
   04 Class-Code PIC S9(4) BINARY.
   04 Cause-Code PIC S9(4) BINARY.
03 Case-Sev-Ctl PIC X.
03 Facility-ID PIC XXX.
02 I-S-Info   PIC S9(9) BINARY.

LINKAGE SECTION.
01 Current-condition.
02 Condition-Token-Value.
COPY CEEIGZCT.
03 Case-1-Condition-ID.
   04 Severity   PIC S9(4) BINARY.
   04 Msg-No     PIC S9(4) BINARY.
03 Case-2-Condition-ID
   REDEFINES Case-1-Condition-ID.
   04 Class-Code PIC S9(4) BINARY.
   04 Cause-Code PIC S9(4) BINARY.
03 Case-Sev-Ctl PIC X.
03 Facility-ID PIC XXX.
02 I-S-Info   PIC S9(9) BINARY.
**
01 Token       PIC X(4).
**
01 Result-code PIC S9(9) BINARY.
   88 resume   VALUE +10.
   88 percolate VALUE +20.
   88 perc-sf  VALUE +21.
   88 promote  VALUE +30.
   88 promote-sf VALUE +31.
01 New-condition.
02 Condition-Token-Value.
COPY CEEIGZCT.
03 Case-1-Condition-ID.
   04 Severity   PIC S9(4) BINARY.
   04 Msg-No     PIC S9(4) BINARY.
03 Case-2-Condition-ID
   REDEFINES Case-1-Condition-ID.
   04 Class-Code PIC S9(4) BINARY.
   04 Cause-Code PIC S9(4) BINARY.
03 Case-Sev-Ctl PIC X.
03 Facility-ID PIC XXX.
02 I-S-Info   PIC S9(9) BINARY.

PROCEDURE DIVISION USING current-condition, token,
   result-code, new-condition.
***********************************************************
** Determine if entry was for OUT OF STORAGE condition. **
***********************************************************
IF CEE0PD OF current-condition THEN
   DISPLAY "COBOL subroutine could NOT RUN because",
   " of the insufficient storage condition."
***********************************************************
** Call CEEMRCR to move the resume cursor           **
***********************************************************
   MOVE 0 TO Movetyp
   CALL "CEEMRCR" USING Movetyp, Feedback
   IF CEE000 OF Feedback THEN
      SET resume TO TRUE
   ELSE
      SET promote TO TRUE
      MOVE feedback TO new-condition
      END-IF
   ELSE
      SET percolate TO TRUE
      END-IF
   GOBACK.
END PROGRAM CECNDHD.
```
When CESUBXP gets control, a request is made to Language Environment to allocate storage for the declared array W2. An out-of-storage condition takes place, provided the caller has not allocated a large amount of storage.

IDENTIFICATION DIVISION.

PROGRAM-ID. CESUBXP.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 ARRAY.

05 W2 PIC X OCCURS 3000000 TIMES.

LINKAGE SECTION.

01 PARM1 PIC X.

08 RAN-OK VALUE "Y".

PROCEDURE DIVISION USING PARM1.

PARA-CND01A.

MOVE "B" TO W2(2999999).

SET RAN-OK TO TRUE.

GOBACK.

End program CESUBXP.

PL/I examples using CEEHDLR, CEEGTST, CEECZST, and CEEMRCR

The following program calls CEEHDLR to register a user-written condition handler for the out-of-storage condition, calls CEEGTST to allocate heap storage, and calls CEECZST to alter the size of the heap storage requested.

*Process macro;
/*****************************/
/* CECNDXP - Call the following Language Environment services: */
/* - CEEHDLR - Register user condition handler */
/* - CEEGTST - Get heap storage */
/* - CEECZST - Change the size of heap element */
/* - CEEMRCU - Unregister user condition handler */
/* 1. A user condition handler CEENDHD is registered. */
/* 2. A large amount of HEAP storage is allocated. */
/* 3. A subroutine, Sub, is called which is known to require a large amount of storage. It is not known whether the storage for Sub is available during this run of the application. */
/* 4. If sufficient storage for Sub is not available, a storage condition is generated. */
/* 5. CECNDXP gets control and sets resume at the next instruction following the call to Sub. */
/* 6. A test for completion of Sub is made after the subroutine call. If Sub did not complete, a large amount of storage is freed, and Sub is invoked a second time. */
/* 7. Sub runs successfully once it has enough storage available. */
/* Note: In order for this example to complete successfully, the FREE suboption of the HEAP runtime option must be in effect. */
/* */
/*****************************/
Cecndxp: proc options(main);

%include ceeibmct;
%include ceeibmaw;
When any condition occurs in CECNDXP, the user condition handler CECNDHD in the following program receives control and tests for the out-of-storage condition. If the out-of-storage condition has occurred, then CECNDHD calls CEEMRCR to return to the instruction in the main program after the subroutine call that produced the out-of-storage condition.

*Process macro;
/*****************************/
Signaling and handling a condition in a C/C++ routine

The next program shows how a user-written condition handler gains control for a condition that was signaled using CEESGL, and calls CEEGQDT to access a data structure that was set up in the signaling routine. The CEEMRCR callable service resets the resume cursor, and execution resumes at the new point.
/* CEESGL   -- signal a condition to the condition manager            */
/* CEEGQDT  -- get the q_data_token                                   */
/* CEEMRCR  -- move the resume cursor                                 */
/*                                                                    */
/* The example also shows how to directly construct a condition token */
/* and provides a sample user condition handler.                      */
 /**********************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

void b(void);
#endif
void handler(_FEEDBACK *,_INT4 *,_INT4 *,_FEEDBACK *);
#endif

typedef struct {          /* condition info structure */
  int   error_value;
  char   err_msg[80];
  int   retcode;
} info_struct;

int main(void) {
  printf("In main program\n");
  b();
  /* CEEMRCR should put the resume cursor at this point */
  printf("Finished\n");
}
void b(void) {
  _FEEDBACK fc,condtok;
  _ENTRY routine;
  _INT4 token,qdata;
  info_struct *info;
  _INT2 c_1,c_2,cond_case,sev,control;
  _CHAR3 facid;
  _INT4 isi;
  printf("In routine b\n");
  token = 99;
  routine.address = (_POINTER)&handler;
  routine.nesting = NULL;
  /* register the condition handler: */
  CEEHDLR(&routine,&token,&fc);
  if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
    printf("CEEHDLR failed with message number %d\n", fc.tok_msgno);
    exit(2999);
  }
  /* build the condition token */
  c_1 = 1;
  c_2 = 99;
  cond_case = 1;
  sev = 1;
  control = 0;
  memcpy(facid,"ZZZ",3);
  isi = 0;
  CEENCOD(&c_1,&c_2,&cond_case,&sev,&control,
           facid,&isi,&condtok,&fc);
  if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
    printf("CEENCOD failed with message number %d\n", fc.tok_msgno);
    exit(2999);
  }
  /* set up the condition info structure */
  info = (info_struct *)malloc(sizeof(info_struct));
  if (info == NULL) {
    printf("error allocating info_struct\n");
    exit(2399);
  }
Handling a divide-by-zero condition in a COBOL program

The following routine illustrates how a COBOL program can handle a divide-by-zero condition if one occurs. These actions occur:

1. The program enables the divide-by-zero exception. Exceptions can be enabled or disabled by calling the CEE3SPM (Query and Modify Language Environment Hardware Condition Enablement) callable service.
2. The program registers a user-written condition handler that recognizes the divide-by-zero condition.
3. The program then performs a divide-by-zero, which causes the user-written condition handler to get control.
4. The handler calls CEE3GRN (Get Name of Routine that Incurred Condition), to return the name of the routine that the condition occurred in.
5. The handler inserts the routine name and condition token into a user-defined message string, and calls CEEMOUT (Dispatch a Message) to send the message to the Language Environment message file.
The Language Environment message file is a file that you can specify to store messages from a given routine or application, or from all routines that run under Language Environment.

6. The program uses CEEHDLR to register the user-written condition handler.

```
CBL LIB,QUOTE,NODYNAM

*** Module/FileName: IGZTSIGR
*************************************************************************
** IGZTSIGR - Call the following Language Environment services: *
** : CEEHDLR - register user condition handler *
** : CEE3GRN - get name of routine that incurred the condition. *
** : CEEMOUT - output message associated with the condition, including the name of the routine that incurred the condition. *
** 1. Our example registers user condition handler IGZTSIGH. *
** 2. Our program then divides by zero, which causes a hardware exception condition. *
** 3. IGZTSIGH gets control and prints out a message that includes the name of the routine that incurred the divide-by-zero condition, IGZTSIGR. *
** 4. IGZTSIGH requests that Condition Management resume execution after the point at which the condition occurred. *
** 5. IGZTSIGR terminates normally. *
*************************************************************************

IDENTIFICATION DIVISION.
PROGRAM-ID.    IGZTSIGR.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  DIVISOR         PIC S9(9) BINARY.
01  QUOTIENT        PIC S9(9) BINARY.
** Declares for condition handling
** 01 PGMPTR     USAGE IS PROCEDURE-POINTER.
01 FBCODE.
  02  Condition-Token-Value.
COPY CEEIGZCT.
  03 Case-1-Condition-ID.
    04 Severity   PIC S9(4) BINARY.
    04 Msg-No     PIC S9(4) BINARY.
  03 Case-2-Condition-ID
    REDEFINES Case-1-Condition-ID.
    04 Class-Code  PIC S9(4) BINARY.
    04 Cause-Code  PIC S9(4) BINARY.
  03 Case-Sev-Ctl  PIC X.
  03 Facility-ID  PIC XXX.
  02 I-S-Info            PIC S9(9) BINARY.
77  TOKEN     PIC X(4).       PROCEDURE DIVISION.
0001-BEGIN-PROCESSING.
  DISPLAY "*********************************".
  DISPLAY "IGZTSIGR COBOL Example is now in motion. ".
  DISPLAY "*********************************".
** Register user condition handler IGZTSIGH **
** using CEEHDLR **
*************************************************************
SET PGMPTR TO ENTRY "IGZTSIGH".
MOVE 97 TO TOKEN.
CALL "CEEHDLR" USING PGMPTR, TOKEN, FBCODE.
IF ( NOT CEE000 of FBCODE ) THEN
  DISPLAY "Error " Msg-No of FBCODE " registering condition handler " IGZTSIGH " UPON CONSOLE.
STOP RUN
```
User-written condition handler

END-IF.

******************************************************************************
** Divide by zero to cause a hardware exception**
** condition.  Condition handler IGZTSIGH gets **
** control and CALLs CEE3GRN to obtain the **
** name of the routine in which the condition **
** was raised. **
** IGZTSIGH then prints a message using CEEMOUT**
** and passing the name "LEASMSIG." Control **
** returns and normal termination takes place. **
** ******************************************************************************

MOVE 0 TO DIVISOR.
DIVIDE 5 BY DIVISOR GIVING QUOTIENT.
DISPLAY "**************************************************************************".
DISPLAY "IGZTSIGR COBOL Example has ended."
DISPLAY "**************************************************************************"
GOBACK.

End program IGZTSIGR.

CBL LIB,QUOTE,NODYNAM

******************************************************************************
** IGZTSIGH - Call the following Language **
** Environment services: **
** **
** : CEE3GRN - Get name of routine that **
** incurred a condition. **
** : CEEMOUT - output a user message **
** **
** This is the user condition handler **
** registered by IGZTSIGR. It calls CEE3GRN **
** to retrieve the name of the routine that **
** incurred the divide-by-zero condition. It **
** then calls CEEMOUT to output the message. **
** ******************************************************************************

IDENTIFICATION DIVISION.
PROGRAM-ID. IGZTSIGH.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01  msgstr.
    02  VarStr-length       PIC S9(4) BINARY.
    02  VarStr-text.
    03  VarStr-char         PIC X,
                 OCCURS 0 TO 256 TIMES
                 DEPENDING ON VarStr-length
                 OF msgstr.
    01  Feedback.
  02  Condition-Token-Value.
    COPY CEEIGZCT.
    03  Case-1-Condition-ID.
        04  Severity    PIC S9(4) BINARY.
        04  Msg-No      PIC S9(4) BINARY.
    03  Case-2-Condition-ID
        REDEFINES Case-1-Condition-ID.
        04  Class-Code  PIC S9(4) BINARY.
        04  Cause-Code  PIC S9(4) BINARY.
    03  Case-Sev-Ctl    PIC X.
    03  Facility-ID     PIC XXX.
    02  I-S-Info            PIC S9(9) BINARY.
    77  rtn-name        PIC X(80).
    77  msgdest         PIC S9(9) BINARY.
    77  string-pointer  PIC S9(4) BINARY.

**
LINKAGE SECTION.
  01  Current-condition.
    COPY CEEIGZCT.
    03  Case-1-Condition-ID.
        04  Severity    PIC S9(4) BINARY.
        04  Msg-No      PIC S9(4) BINARY.
    03  Case-2-Condition-ID
        REDEFINES Case-1-Condition-ID.
        04  Class-Code  PIC S9(4) BINARY.
        04  Cause-Code  PIC S9(4) BINARY.
    03  Case-Sev-Ctl    PIC X.
    03  Facility-ID     PIC XXX.
    02  I-S-Info            PIC S9(9) BINARY.

**
  01  Token           PIC X(4).
  **
  01  Result-code     PIC S9(9) BINARY.
    88 resume     VALUE +10.
    88 percolate  VALUE +20.

Coding a user-written condition handler 227
PROCEDURE DIVISION USING current-condition, token, result-code, new-condition.

*************************************************
*    Check to see whether this routine was      *
*    entered due to a divide-by-zero exception, *
*    or due to some other condition.            *
*************************************************
IF CEE349 OF current-condition THEN

*************************************************
*      (A divide-by-zero condition has occurred)*
*************************************************
SET resume TO TRUE

*************************************************
**   Call CEE3GRN to retrieve the name of the  **
**   program that incurred the divide-by-zero  **
**   exception. Build user message and include **
**   the name of the program.                   **
*************************************************
CALL "CEE3GRN" USING rtn-name, feedback
IF ( NOT CEE000 OF feedback ) THEN
DISPLAY "Error " Msg-No OF feedback
" in obtaining program name."
UPON CONSOLE
MOVE feedback TO new-condition
SET promote TO TRUE
ELSE
MOVE 1 TO string-pointer
MOVE 255 TO VarStr-length OF msgstr
STRING "The example program "
  rtn-name
  " incurred a divide-by-zero"
  " exception."
DELIMITED BY "  "
INTO VarStr-text OF msgstr
POINTER string-pointer
SUBTRACT 1 FROM string-pointer,
GIVING VarStr-length OF msgstr
MOVE 2 TO msgdest

*************************************************
**     Call CEEMOUT to output the user message.**
*************************************************
CALL "CEEMOUT" USING msgstr,msgdest, feedback
IF ( NOT CEE000 OF feedback ) THEN
DISPLAY "Error in writing the "
"message string."
MOVE feedback TO new-condition
SET promote TO TRUE
END-IF
ELSE

*************************************************
*        (A condition other than               *
*         divide-by-zero has occurred)         *
*************************************************
SET percolate TO TRUE
END-IF
GOBACK.

END PROGRAM IGZTSIGH.
Handling a program check in an assembler routine

The following routine illustrates how an assembler routine can handle a program check if one should occur. The following occurs:

1. The routine registers a user-written condition handler, LEASMHD3, that responds to a program check by calling CEE3DMP to request a dump.
2. The routine then calls a subroutine, LEASMHD2, that generates a program check.
3. The routine gives control to the user-written condition handler.

Note that a condition handler to which an assembler routine gives control does not have to be link-edited into the same load module as the routine; a condition handler can be dynamically loaded and can possibly dynamically load other modules also.

```plaintext
SMP1     TITLE 'Sample of main program that registers a handler'
*     Symbolic Register Definitions and Usage
* R0    EQU   0             Parameter list address (CMS only)
R1    EQU   1             Parameter list address, 0 if no parms
R10   EQU   10            Base register for executable code
R12   EQU   12            Language Environment Common Anchor Area
*             address
R13   EQU   13            Save Area/Dynamic Storage Area address
R14   EQU   14            Return point address
R15   EQU   15            Entry point address
*     Prologue
* CEEHDRA CEEENTRY AUTO=DSASIZ,     Amount of main memory to obtain
    MAIN=YES,          This routine is a MAIN program
    PPA=PPA1,          Program Prolog Area for this routine
    BASE=R10           Base register for executable code
*            constants, and static variables
* USING CEECAA,R12         Common Anchor Area addressability
* USING CEEDSA,R13         Dynamic Storage Area addressability
*     Announce ourselves
* WTO  'CEEHDRA Says "HELLO"',ROUTCDE=11
*     Register User Handler
* LA    R1,USRHDLPP        Get addr of proc-ptr to Handler rtn
ST    R1,PARM1           Make it 1st parameter
LA    R1,TOKEN           Get addr of 32-bit token
ST    R1,PARM2           Make it 2nd parameter
LA    R1,0               Omit address for Feedback Code:
*             If an error occurs while
*             registering the handler,
*             Language Environment signals
*             the condition, rather than
*             passing it back to caller
* ST    R1,PARM3           Make it 3rd parameter
LA    R1,HDLRPLST        Point to parameter list for CEEHDLR
CALL  CEEHDLR            Invoke CEEHDLR callable service AWI
*     Call subroutine to cause an exception
* CALL  LEASMHD2
*     Un-Register User Handler
* LA    R1,USRHDLPP        Get addr of proc-ptr to Handler rtn
ST    R1,HDLUPRM1        Make it 1st parameter
LA    R1,FEEDBACK        Address for Feedback Code
* ST    R1,HDLUPRM2        Make it 2nd parameter
LA    R1,HDLRPLST        Point to parameter list for CEEHDLR
CALL  CEEHDLU            Invoke CEEHDLU callable service AWI*
* Bid fond farewell
* WTO  'CEEHDRA Says "GOOD-BYE"',ROUTCDE=11
*     Epilogue
* CEETERM RC=4,MODIFIER=1  Terminate program
```
* Program Constants and Local Static Variables
* USRHDPP DC V(LEASMHD3),A(0) Procedure-pointer to Handler routine
* LTORG , Place literal pool here
PPA1 CEEPPA , Program Prolog Area for this routine
EJECT
* Map the Dynamic Storage Area (DSA)
* CEEDSA , Map standard CEE DSA prologue
* Local Automatic (Dynamic) Storage..
* HDLRPLST DS 0F Parameter List for CEEHDLR
PARM1 DS A Address of User-written Handler
PARM2 DS A Address of 32-bit Token
PARM3 DS A Address of Feedback Code cond token
* HDLUPRM1 DS A Address of User-written Handler
HDLUPRM2 DS A Address of Feedback Code cond token
* TOKEN DS F 32-bit Token: fullword whose *value* will be passed to the user handler each time it is called.
* FEEDBACK DS CL12 Feedback Code condition token
* DSASIZ EQU *-CEEDSA Length of DSA
EJECT
* Map the Common Anchor Area (CAA)
* CEECAA END CEEHDRA
HDR2 TITLE 'Sample of subprogram that forces a program check'
* Symbolic Register Definitions and Usage
* R1 EQU 1 Parameter list address, 0 if no parms
R11 EQU 11 Base register for executable code
R12 EQU 12 Language Environment Common Anchor Area address
* R13 EQU 13 Save Area/Dynamic Storage Area address
R14 EQU 14 Return point address
R15 EQU 15 Entry point address*
* Prologue
* LEASMHD2 CEEENTRY AUTO=DSASIZ, Amount of main memory to obtain
PPA=PPA2, Program Prolog Area for this routine
MAIN=NO, This program is a Subroutine
NAB=YES, YES because called by enabled rtn
BASE=R11 Base register for executable code, constants, and static variables
* USING CEECAA,R12 Common Anchor Area addressability
USING CEEDSA,R13 Dynamic Storage Area addressability
* Announce ourselves
* WTO 'LEASMHD2 Says "HELLO"', ROUTCDE=11
* Cause Data Exception (Language Environment condition 3207)
* XC A,A Clear to Binary Zeros
* (not a valid packed number)
* AP A,=P'7' Cause Data exception
* Say good-bye
* WTO 'LEASMHD2 Says "GOOD-BYE"', ROUTCDE=11
* Epilogue
* CEETERM RC=0 Terminate program
SPACE 3
* Program Constants and Local Static Variables
* PPA2 CEEPPA , Program Prolog Area for this routine
* LTORG , Place literal pool here
User-written condition handler

Coding a user-written condition handler 231

EJECT
*  Map the Dynamic Storage Area (CAA)
*  CEEDSA , Map standard CEE DSA prologue
*  Local Automatic (Dynamic) Storage..
A DS PL2 Packed operand (uninitialized)
* DSASIZ EQU *-CEEDSA Length of DSA
EJECT
*  Map the Common Anchor Area (CAA)
*  CEECAA
END , of LEASMHD2
SMP3 TITLE 'User-written condition handler'*
* Symbolic Register Definitions and Usage
*  R1 EQU 1 Parameter list address (upon entry)
  R2 EQU 2 Work register
  R3 EQU 3 Parameter list address (after CEEENTRY)
  R4 EQU 4 Will point to Result Code Argument
  R10 EQU 10 Will point to Condition Token Argument
  R11 EQU 11 Base register for executable code
  R12 EQU 12 Common Anchor Area address
  R13 EQU 13 Save Area/Dynamic Storage Area address
  R14 EQU 14 Return point address
  R15 EQU 15 Entry point address
*  Prologue
*  LEASMHD3 CEEENTRY AUTO=DSASIZ, Amount of main memory to obtain *
   PPA=PPA3, Program Prolog Area for this routine *
   MAIN=NO, This program is a Subroutine *
   NAB=YES, YES--called under Language Env. *
   PARMREG=R3, R1 value is saved here *
   BASE=R11 Base register for executable code,
   constants, and static variables
   USING CEECAA,R12 Common Anchor Area addressability
   USING CEEDSA,R13 Dynamic Storage Area addressability
   USING UHDLARGS,R3 User Handler Args addressability
*  Locate Arguments
*   L R10,@CURCOND Get address of Condition Token
   USING $CURCOND,R10 Condition Token addressability
   L R4,@RESCODE Get address of Result Code
   USING $RESCODE,R4 Result Code addressability
*  Announce ourselves
*   WTO 'LEASMHD3 Says "HELLO"',ROUTCDE=11
*  Process Condition
*   CLC CURCOND(B),CEE347 Was this handler entered due to the *
   condition it was created to deal with (data exception) ?
   BE BADPDATA Yes -- go process it No..
   MVC RESCODE,A(PERCOLAT) Indicate PERCOLATE action
   B OUT Return to Language Environment condition manager
*  BADPDATA EQU * Processing for data exception:
   MVC RESCODE,A(RESUME) Indicate RESUME action*
   Call CEE3DMP to Dump machine state
*  LA R1,DUMPTITL Get address of Dump Title
  ST R1,PARM1 Make it first parameter
  LA R1,DUMPOPTS Get address of Dump Options string
  ST R1,PARM2 Make it second parameter
  LA R1,FC Address of Feedback Code
  ST R1,PARM3 Make it third parameter
  LA R1,DMPPARAMS Point to parameter list for CEE3DMP
  CALL CEE3DMP Invoke CEE3DMP callable service AwI
*  Sign-off
*  OUT EQU *
User-written condition handler

\[
\text{WTO 'LEASMHD3 Says "GOOD-BYE"', ROUTCODE=11}
\]
\[
\text{* Epilogue}
\]
\[
\text{* CEEETERM RC=0}
\]
\[
\text{* Program Constants and Local Static Variables}
\]
\[
\text{* DUMPOPTS DC CL256'THR(ALL) BLOCK STORAGE' Dump Options}
\]
\[
\text{* DUMPTITL DC CL80'LEASMHD3 - Sample Dump ' Dump Title}
\]
\[
\text{* PPA3 CEEPPA, Program Prolog Area for this routine}
\]
\[
\text{* LTORG, Place literal pool here}
\]
\[
\text{* Define Symbolic Value Constants for Condition Tokens}
\]
\[
\text{CEEBALCT}
\]
\[
\text{EJECT}
\]
\[
\text{* Map Arguments to User-Written Condition Handler}
\]
\[
\text{UHDLARGS DSECT}
\]
\[
\text{@CURCOND DS A Address of CIB}
\]
\[
\text{@TOKEN DS A Address of 32-bit token value from CEEHDLR}
\]
\[
\text{@RESCODE DS A Address of Result Code}
\]
\[
\text{@NEWCOND DS A Address of New Condition}
\]
\[
\text{SPACE 3}
\]
\[
\text{$CURCOND DSECT , Mapping of the current condition}
\]
\[
\text{CURCOND DS A Condition token that identifies the}
\]
\[
\text{current condition being processed}
\]
\[
\text{SPACE 3}
\]
\[
\text{$TOKEN DSECT , Mapping of the 32-bit Token Argument}
\]
\[
\text{TOKEN DS A Value of 32-bit Token from CEEHDLR call}
\]
\[
\text{SPACE 3}
\]
\[
\text{$RESCODE DSECT , Mapping of Result Code Argument}
\]
\[
\text{RESCODE DS F Result Code specifies the action for}
\]
\[
\text{the condition manager to take when control returns from the user handler:}
\]
\[
\text{RESUME EQU 10 Resume at the resume cursor}
\]
\[
\text{PERCOLAT EQU 20 Percolate to the next condition handler}
\]
\[
\text{PROMOTE EQU 30 Promote to the next condition handler}
\]
\[
\text{*(See the Language Environment Programming Guide for other result}
\]
\[
\text{code values.)}
\]
\[
\text{SPACE 3}
\]
\[
\text{$NEWCOND DSECT , Mapping of the New Condition Argument}
\]
\[
\text{NEWCOND DS CL12 New Condition (condition token) specifies}
\]
\[
\text{the condition promoted to.}
\]
\[
\text{EJECT}
\]
\[
\text{* Map the Dynamic Storage Area (DSA)}
\]
\[
\text{CEEDSA , Map standard CEE DSA prologue}
\]
\[
\text{* Local Automatic (Dynamic) Storage...}
\]
\[
\text{DMPPARMS DS GF Parameter list for CEE3DMP}
\]
\[
\text{PARM1 DS A Address of Title string}
\]
\[
\text{PARM2 DS A Address of Options string}
\]
\[
\text{PARM3 DS A Address of Feedback Code}
\]
\[
\text{* FC DS CL12 Feedback Code condition token}
\]
\[
\text{* DSASIZ EQU *-CEEDSA Length of DSA}
\]
\[
\text{EJECT}
\]
\[
\text{* Map the Common Anchor Area (CAA)}
\]
\[
\text{CEECAA END , of LEASMHD3}
\]
Chapter 18. Using condition tokens

Language Environment uses the 12-byte condition token data type to perform a variety of communication functions. This topic 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.

The basics of using condition tokens

If you provide an \texttt{fc} parameter in a call to a Language Environment callable service, the service sets \texttt{fc} to a specific value called a condition token and returns it to your application. (See “The effect of coding the \texttt{fc} parameter” on page 233 for more information.)

If you do not specify the \texttt{fc} parameter in a call to a Language Environment service, Language Environment generates a condition token for any nonzero condition and signals it using the CEESGL callable service. Signaling the condition token causes it to be passed it to Language Environment condition handling. (See “Effects of omitting the \texttt{fc} parameter” on page 235 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. See “User-written condition handler interface” on page 204 for more information about coding user-written condition handlers. The structure of the condition token is described in “Understanding the structure of the condition token” on page 235, and symbolic feedback codes are discussed in “Using symbolic feedback codes” on page 236.

Language Environment condition tokens contain a 4-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 condition handlers use to identify and react to a specific condition.

Language Environment provides callable services to help you construct and decompose your own condition tokens.

\texttt{CEEDCOD}

Breaks down a condition token into its component parts.

\texttt{CEENCOD}

Creates a new condition token in your application.

See \textit{z/OS Language Environment Programming Reference} for a detailed explanation of each field in a condition token and for more information about using CEEDCOD and CEENCOD callable services. See also the message handling services listed in Chapter 19, "Using and handling messages," on page 257.

The effect of coding the \texttt{fc} parameter

The feedback code is the last parameter of all Language Environment callable services, and the second to last parameter of all Language Environment math services. COBOL/370 programs must provide the feedback code parameter in each call to a Language Environment callable service; C, C++, Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, and PL/I routines do not have to do so. (See \textit{z/OS Language Environment Programming Reference} for information about how to provide the feedback code parameter in each HLL.) When the \texttt{fc} parameter is provided and a condition is raised, the following sequence of events occurs:
1. The callable service in which the condition occurred builds a condition token for the condition. The condition token is a 12-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 condition

   For example, you can use the CEEBLDTX utility (see “Creating messages” on page 257) or the CEECMI callable service (see z/OS Language Environment Programming Reference) to generate message inserts. Routines signaling a new condition with a call to CEESGL should first call CEECMI to copy any insert information into the ISI associated with the condition.

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 172.

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.
   - Signal it to Language Environment using the CEESGL callable service.
   - Get, format, and dispatch the message for display using the CEEMSG callable service.
   - Store the message in a storage area using the CEEMGET callable service.
   - Use the CEEMOUT 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 12-byte condition token containing all zeroes coded in your routine. See “Using symbolic feedback codes” on page 236 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, or you can call the CEEDCOD (decompose condition token) service to break down the condition token. See z/OS Language Environment Programming Reference for more information about the CEEDCOD service.

- 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 12-byte condition token. See “Using symbolic feedback codes” on page 236 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 12 bytes of each condition token with each other. The last 4 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 12-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 172. If the condition remains unhandled at the end of processing, Language Environment takes the Language Environment default action (defined in Table 34 on page 174). 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 74 on page 235 illustrates the structure of the condition token, with bit offsets shown above the components:

```
  0 - 31 32-33 34 - 36 37 - 39 40 - 63 64 - 95
  |   |   |   |   |   |   |
  | Condition_ID | Case Number | Severity Number | Control Code | Facility_ID | ISI |
```

For Case 1 condition tokens, Condition_ID is:

```
  0 - 15 16 - 31
  |   |   |
  | Severity Number | Message Number |
```

For Case 2 condition tokens, Condition_ID is:

```
  0 - 15 16 - 31
  |   |   |
  | Class Code | 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 74: Language Environment condition token
Every condition token contains the components indicated in Figure 74 on page 235.

**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 74 on page 235 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. The facility ID for PL/I, for example, is IBM.

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 are creating a new facility ID to use with your own message source file, follow the guidelines listed under the Facility_ID parameter of CEENCOD in z/OS Language Environment Programming Reference.

If you create a new facility_ID to use with a message source file you created using CEEBLDTX (see “Creating messages” on page 257), 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**
A 4-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 4 bytes of qualifying data. The ISI token is typically built by Language Environment for system or Language Environment-signaled conditions. It can also be built by an application for conditions signaled using CEEESGL. The CEECMI callable service can be used to define the message inserts within the ISI for a condition token. The q_data to be placed in the ISI for a condition token is defined by signaling the condition using CEEESGL.

You can extract ISI information inside of CEEHDLR-established condition handlers. 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 CEEMGET. The q_data_token can be retrieved using CEEGQDT.

**Using symbolic feedback codes**
Language Environment provides symbolic feedback codes representing the first 8 bytes of a 12-byte condition token. Using Language Environment 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” on page 237 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. The symbolic feedback code table for the CEEGTST (get heap storage) callable service is shown in Table 42 on page 237.

<table>
<thead>
<tr>
<th>Symbolic feedback code</th>
<th>Severity</th>
<th>Message number</th>
<th>Message text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE000</td>
<td>0</td>
<td>—</td>
<td>The service completed successfully.</td>
</tr>
<tr>
<td>CEE0P2</td>
<td>4</td>
<td>0802</td>
<td>Heap storage control information was damaged.</td>
</tr>
<tr>
<td>CEE0P3</td>
<td>3</td>
<td>0803</td>
<td>The heap identifier in a get storage request or a discard heap request was unrecognized.</td>
</tr>
<tr>
<td>CEE0P8</td>
<td>3</td>
<td>0808</td>
<td>Storage size in a get storage request or a reallocate request was not a positive number.</td>
</tr>
<tr>
<td>CEE0PD</td>
<td>3</td>
<td>0813</td>
<td>Insufficient storage was available to satisfy a get storage request.</td>
</tr>
</tbody>
</table>

To test for the condition raised when you specify an invalid heap ID from which to get storage, you can compare the symbolic feedback code CEE0P3 to the condition token returned either from the service or from the Language Environment condition manager (depending on whether you specified fc in the call to CEEGTST).

- If you want to code a condition handling routine to handle a condition resulting in an error message from your application, see z/OS Language Environment Runtime Messages, which lists error messages and the symbolic feedback code for conditions.

Including symbolic feedback code files

Symbolic feedback codes are provided for Language Environment, C or C++, COBOL, Fortran, and PL/I 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).

- **yyy**
  Indicates the facility ID of the language in which the declarations are coded. For example, EDCIBMCT contains PL/I declarations of C condition tokens. yyy can be BAL (assembler), EDC (C or C++), FOR (Fortran), IBM (PL/I), or IGZ (COBOL).

- **CT**
  Stands for “condition token.”

To use symbolic feedback codes, include the file in your source code using the appropriate language construct, for example:
Using condition tokens

- In C or C++, to include the file of C or C++ declarations for IGZ (COBOL) condition tokens, specify:
  
  ```
#include <igzedcct>
  ```

- In COBOL, define SCEESAMP and use the COPY statement to include the file, as shown below.
  Define SCEESAMP in your SYSLIB statement:

  ```
//SYSLIB DD DSNAME=CCE.SCEESAMP,DISP=SHR
  ```

  Specify the following in your COBOL code to include the files containing Language Environment and COBOL condition tokens declared in COBOL:

  ```
COPY CEEIGZCT.
COPY IGZIGZCT.
  ```

- In Fortran, to include the Fortran declarations for FOR (Fortran) and CEE (Language Environment) condition tokens, specify the following.

  ```
INCLUDE (FORFORCT)
INCLUDE (CEEFORCT)
  ```

- In PL/I, to include the PL/I declarations for IBM (PL/I) and CEE (Language Environment) condition tokens, specify:

  ```
%INCLUDE IBMIBMCT
%INCLUDE CEEIBMCT
  ```

Examples using symbolic feedback codes

The following examples use symbolic feedback codes to test user input and display a message if the input is incorrect.

**C and C++**

In the following example, the symbolic feedback code file CEEEDCCT is included and a call is made to CEEGTST. After the call, a test is made for the condition token representing an invalid heap ID. The \textit{fc} returned from CEEGTST is tested against the symbolic feedback code CEE0P3 listed in the CEEGTST feedback code table (see \textit{z/OS Language Environment Programming Reference}). If the heap ID specified is not valid, another call is made to CEEGTST to try again.

\texttt{_FBCHECK} (IBM-supplied) is used to compare only the first 8 bytes of the \textit{fc} against the symbolic feedback code.
/*Module/File Name:  EDCSFC   */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

main(void)
{
    _FEEDBACK fc;
    _POINTER address;
    _INT4 heapid, size;

    size = 1000;
    heapid = 999;

    CEEGTST(&heapid, &size, &address, &fc);
    if ((_FBCHECK (fc, CEE0P3)) == 0){
        printf("You specified a Heap Id that does not exist!\n\n");
    }
    printf("Try again:\n");
    heapid = 0;

    CEEGTST(&heapid, &size, &address, &fc);
    if ((_FBCHECK (fc, CEE000)) == 0){
        printf("Now it worked!\n");
    } else {
        printf("CEEGTST failed with message number%d \n", fc.tok_msgno);
        exit(99);
    }

    return 0;
}

Figure 75: C/C++ example testing for CEEGTST symbolic feedback code CEE0P3

COBOL
In Figure 76 on page 240, the symbolic feedback code file CEEIGZCT is accessed and a call is made to
CEESDEXP (exponential base e). The first 8 bytes of the feedback code returned are tested against the
symbolic feedback code CEE1UR to ensure that the input parameter is within the valid range for
CEESDEXP. The symbolic feedback code table for CEESDEXP is listed in z/OS Language Environment
Programming Reference. A message is displayed if the input parameter is out of range.
Using condition tokens

Figure 76: COBOL example testing for CEESDEXP symbolic feedback code CEE1UR

It is important that symbolic feedback codes be compared with only the first 8 bytes of the 12-byte condition token. To this end, you must code the COPY statements for the symbolic feedback code declarations in the right place within the condition token declaration.

In Figure 76 on page 240, for example, symbolic feedback code CEE1UR is compared to the first 8 bytes of condition token FBC because of the correct placement of the COPY statements.

It is wrong to place the COPY statements before the declaration of Condition-Token-Value as shown in Figure 77 on page 241, because the 8-byte symbolic feedback code blank-padded (X'40') to a length of 12 bytes would be compared to the full 12-byte condition token. The comparison would always fail, because the blanks would not match the ISI data in the last 4 bytes of the condition token.
```
01 FBC
   COPY CEEIGZCT.          <<+++ Incorrect
   COPY IGZIGZCT.          <<+++ Incorrect
   02 Condition-Token-Value
      03 Case-1-Condition-ID.
         04 Severity  PIC S9(4) BINARY.
         04 Msg-No    PIC S9(4) BINARY.
      03 Case-2-Condition-ID
         REDEFINES Case-1-Condition-ID.
         04 Class-Code PIC S9(4) BINARY.
         04 Cause-Code PIC S9(4) BINARY.
      03 Case-Sev-Ctl  PIC X.
      03 Facility-ID  PIC XXX.
   02 I-S-Info       PIC S9(9) BINARY.
```

*Figure 77: Wrong placement of COBOL COPY statements for testing feedback code*

**PL/I**

The example in Figure 78 on page 242 includes the symbolic feedback code file CEEIBMCT so that Language Environment feedback codes (with facility ID CEE) will be defined. FBCHECK (IBM-supplied) is called to compare the first 8 bytes of FC with the symbolic feedback code CEE000 to determine if the call to CEEMGET is successful. If it is, the message associated with feedback code CEE001 is printed.
**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 43 on page 243 contains condition tokens for C signals seen in C or C++ applications running in a POSIX(OFF) environment. The signals listed in Table 43 on page 243 have a condition token representation with facility ID of EDC.
Table 43: Language Environment condition tokens and non-POSIX C signals

<table>
<thead>
<tr>
<th>Severity</th>
<th>Message number</th>
<th>Symbolic feedback code</th>
<th>Case</th>
<th>Severity</th>
<th>Control</th>
<th>ID</th>
<th>Signal name</th>
<th>Signal number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6000</td>
<td>EDC5RG</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>EDC</td>
<td>SIGFPE</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>6001</td>
<td>EDC5RH</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>EDC</td>
<td>SIGILL</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6002</td>
<td>EDC5RI</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>EDC</td>
<td>SIGSEGV</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>6003</td>
<td>EDC5 RJ</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>EDC</td>
<td>SIGABND</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>6004</td>
<td>EDC5 RK</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>EDC</td>
<td>SIGTERM</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>6005</td>
<td>EDC5 RL</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>EDC</td>
<td>SIGINT</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>6006</td>
<td>EDC5 RM</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>EDC</td>
<td>SIGABRT</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6007</td>
<td>EDC5 RN</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>EDC</td>
<td>SIGUSR1</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>6008</td>
<td>EDC5 RO</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>EDC</td>
<td>SIGUSR2</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>6009</td>
<td>EDC5 RP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>EDC</td>
<td>SIGIOERR</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 44 on page 243 contains condition token for C signals seen in C or C++ applications running in a POSIX(ON) environment. The signals listed in Table 44 on page 243 have a condition token representation with facility ID of CEE.

Table 44: Language Environment condition tokens and POSIX C signals

<table>
<thead>
<tr>
<th>Severity</th>
<th>Message number</th>
<th>Symbolic feedback code</th>
<th>Case</th>
<th>Severity</th>
<th>Control</th>
<th>ID</th>
<th>Signal name</th>
<th>Signal number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5201</td>
<td>CEE52H</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGFPE</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>5202</td>
<td>CEE52I</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGILL</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5203</td>
<td>CEE52 J</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGSEGV</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>5204</td>
<td>CEE52 K</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGABND</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>5205</td>
<td>CEE52 L</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGTERM</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>5206</td>
<td>CEE52 M</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGINT</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5207</td>
<td>CEE52 N</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>CEE</td>
<td>SIGABRT</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5208</td>
<td>CEE52 O</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGUSR1</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>5209</td>
<td>CEE52 P</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGUSR2</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>5210</td>
<td>CEE52 Q</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGHUP</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5211</td>
<td>CEE52 R</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGSTOP</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5212</td>
<td>CEE52 S</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGKILL</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>5213</td>
<td>CEE52 T</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGPIPE</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>5214</td>
<td>CEE52 U</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGALRM</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>5215</td>
<td>CEE52 V</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGCONT</td>
<td>19</td>
</tr>
<tr>
<td>1</td>
<td>5216</td>
<td>CEE530</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGCHLD</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>5217</td>
<td>CEE531</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGTTIN</td>
<td>21</td>
</tr>
</tbody>
</table>
### Table 44: Language Environment condition tokens and POSIX C signals (continued)

<table>
<thead>
<tr>
<th>Severity</th>
<th>Message number</th>
<th>Symbolic feedback code</th>
<th>Case</th>
<th>Severity</th>
<th>Control</th>
<th>ID</th>
<th>Signal name</th>
<th>Signal number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5218</td>
<td>CEE532</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGTTOU</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>5219</td>
<td>CEE533</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGIO</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>5220</td>
<td>CEE534</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGQUIT</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>5221</td>
<td>CEE535</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGTSTP</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>5222</td>
<td>CEE536</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGTRAP</td>
<td>26</td>
</tr>
<tr>
<td>1</td>
<td>5223</td>
<td>CEE537</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGIOERR</td>
<td>27</td>
</tr>
<tr>
<td>1</td>
<td>5224</td>
<td>CEE538</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGDCE</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>5225</td>
<td>CEE539</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGPOLL</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5226</td>
<td>CEE53A</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGURG</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5227</td>
<td>CEE53B</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGBUS</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>5228</td>
<td>CEE53C</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGSYS</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>5229</td>
<td>CEE53D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGWINCH</td>
<td>28</td>
</tr>
<tr>
<td>1</td>
<td>5230</td>
<td>CEE53E</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGXCPU</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>5231</td>
<td>CEE53F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGXFSZ</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>5232</td>
<td>CEE53G</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGVTALRM</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>5233</td>
<td>CEE53H</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>CEE</td>
<td>SIGPROF</td>
<td>32</td>
</tr>
<tr>
<td>5234</td>
<td>CEE53I</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGDUMP</td>
<td>39</td>
</tr>
<tr>
<td>5235</td>
<td>CEE53J</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGDANGER</td>
<td>33</td>
</tr>
<tr>
<td>5236</td>
<td>CEE53K</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGTHSTOP</td>
<td>34</td>
</tr>
<tr>
<td>5237</td>
<td>CEE53L</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CEE</td>
<td>SIGTHCONT</td>
<td>35</td>
</tr>
</tbody>
</table>

**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 can be retrieved using the CEEGQDT callable service from within a CEEHDLR-established condition handler; see “Example illustrating retrieval of q_data” on page 245 for an example invocation, and z/OS Language Environment Programming Reference for syntax of the CEEGQDT service. From a Fortran routine, you can use the Fortran-specific callable services and functions described in Language Environment for MVS & VM Fortran Run-Time Migration Guide to retrieve the q_data, and you do not need to use the q_data_token. The q_data associated with abends is also listed by message number in z/OS Language Environment Runtime Messages.

q_data is comprised of a list of addresses pointing to information that can be used by HLL and user-written condition handlers to react to a condition. The q_data structure for an abend is shown in Figure 79 on page 245.

If an abend occurs, Language Environment signals condition CEE35I (corresponding to message number 3250) and builds the q_data structure shown in Figure 79 on page 245.
Figure 79: 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:

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 30</th>
<th>Bit 29</th>
<th>Bit 28</th>
<th>Bit 27</th>
<th>Bit 26</th>
<th>Bit 25</th>
<th>Bit 24</th>
<th>Bit 23</th>
<th>Bit 22</th>
<th>Bit 21</th>
<th>Bit 20</th>
<th>Bit 19</th>
<th>Bit 18</th>
<th>Bit 17</th>
<th>Bit 16</th>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>8</td>
<td>19</td>
<td>20</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 (as occurs, for example, in a CICS abend), *reason code* has the value zero.

**Usage notes**
- You can use the CEEQDT callable service to retrieve the q_data_token; see *z/OS Language Environment Programming Reference* for more information.
- From a Fortran routine, you can retrieve the qualifying data using Fortran-specific callable services and functions, which are described in *Language Environment for MVS & VM Fortran Run-Time Migration Guide*.

**Example illustrating retrieval of q_data**
The following example shows how the abend code can be retrieved from q_data by invoking the CEEQDT callable service within a CEEHDLR-established condition handler written in COBOL. For an example of a working program that includes the following code, see member IGZTCHDL in library CEE.SCEESAMP.

```cobol
ID DIVISION.
PROGRAM-ID. GETQDATA.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
```
Using condition tokens

*****************************************************************
*       When Language Environment fields a system or user       *
*       abend, condition CEE35I (corresponding to message       *
*       number 3250) is raised. The following code uses       *
*       callable service CEEGQDT to get the q_data and examine   *
*       the abend code.                                        *
*****************************************************************
WHEN CEE35I OF CURRENT-CONDITION
PERFORM
*****************************************************************
*           Get q_data for the condition we are handling.       *
*****************************************************************
CALL "CEEGQDT" USING CURRENT-CONDITION
`q_data` structure for arithmetic program interruptions

If one of the arithmetic program interruptions shown in Table 45 on page 247 occurs, and the corresponding condition is signaled, Language Environment builds the `q_data` structure shown in Figure 80 on page 248.

<table>
<thead>
<tr>
<th>Program interruption (see notes 1 and 2)</th>
<th>Program interruption code</th>
<th>Condition</th>
<th>Message number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point overflow exception</td>
<td>08</td>
<td>CEE348</td>
<td>3208</td>
</tr>
<tr>
<td>Fixed-point divide exception</td>
<td>09</td>
<td>CEE349</td>
<td>3209</td>
</tr>
<tr>
<td>Exponent-overflow exception</td>
<td>0C</td>
<td>CEE34C</td>
<td>3212</td>
</tr>
<tr>
<td>Exponent-underflow exception</td>
<td>0D</td>
<td>CEE34D</td>
<td>3213</td>
</tr>
<tr>
<td>Floating-point divide exception</td>
<td>0F</td>
<td>CEE34F</td>
<td>3215</td>
</tr>
<tr>
<td>Unnormalized-operand exception</td>
<td>1E</td>
<td>CEE34U</td>
<td>3230</td>
</tr>
</tbody>
</table>

Notes:

1. The square root exception is also an arithmetic program interruption, but is treated like the condition from the square root mathematical routine.
2. An arithmetic program interruption that occurs on a vector instruction is presented to a user-written condition handler in the same form as though it had occurred on a scalar instruction. A single vector instruction could cause multiple, possibly different, program interruptions to occur, but each interruption is presented individually.
The q_data structure shown in Figure 80 on page 248 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:

**q_data_token (input)**
The 4-byte address of the address list. This value is returned by the CEEGQDT callable service.

**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. (See “Format of q_data descriptors” on page 254 for information about q_data descriptors.)

**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):

<table>
<thead>
<tr>
<th>Program interruption</th>
<th>Length and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point overflow exception</td>
<td>4- or 8-byte binary integer</td>
</tr>
<tr>
<td>Fixed-point divide exception</td>
<td>8-byte binary integer</td>
</tr>
<tr>
<td>Exponent-overflow exception</td>
<td>4-, 8-, or 16-byte floating-point number</td>
</tr>
<tr>
<td>Exponent-underflow exception</td>
<td>4-, 8-, or 16-byte floating-point number</td>
</tr>
<tr>
<td>Floating-point divide exception</td>
<td>4-, 8-, or 16-byte floating-point number</td>
</tr>
<tr>
<td>Unnormalized-operand exception (occurs only on vector instructions)</td>
<td>4- or 8-byte floating-point number</td>
</tr>
</tbody>
</table>
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 when the user condition handler requests the fix-up and resume action (result code 60 with a condition token of CEE0CF). 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):

<table>
<thead>
<tr>
<th>Program interruption</th>
<th>Length and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point overflow exception</td>
<td>4- or 8-byte binary integer</td>
</tr>
<tr>
<td>Fixed-point divide exception</td>
<td>8-byte binary integer or two 4-byte binary integers (remainder, quotient)</td>
</tr>
<tr>
<td>Exponent-overflow exception</td>
<td>4-, 8-, or 16-byte floating-point number</td>
</tr>
<tr>
<td>Exponent-underflow exception</td>
<td>4-, 8-, or 16-byte floating-point number</td>
</tr>
<tr>
<td>Floating-point divide exception</td>
<td>4-, 8-, or 16-byte floating-point number</td>
</tr>
<tr>
<td>Unnormalized-operand exception (occurs only on vector instructions)</td>
<td>4- or 8-byte floating-point number</td>
</tr>
</tbody>
</table>

**mach_inst_address (input)**
The address of the machine instruction causing the program interruption.

**Usage notes**

- You can use the CEEGQDT callable service to retrieve the q_data_token; see *z/OS Language Environment Programming Reference* for more information.
- From a Fortran routine, you can retrieve the qualifying data using Fortran-specific callable services and functions, which are described in *Language Environment for MVS & VM Fortran Run-Time Migration Guide*.
- Using the q_data structure, a user condition handler can resume either with:
  - The resume action (result code 10) using the value in mach_inst_result. The effect is the same as though execution had continued without any change to the register contents left by the machine instruction.
  - The fix-up and resume action (result code 60 with a condition token of CEE0CF) for exceptions other than unnormalized-operand. This allows any value to be placed in the result register that the machine instruction used.
- You can use the CEE3SPM callable service to set or reset the exponent-underflow mask bit in the program mask; the bit controls whether a program interruption occurs when exponent-underflow occurs, as follows:
  - When the bit is on, the program interruption occurs and condition CEE34D is signaled.
  - When the bit is off, no program interruption occurs; therefore no condition is signaled.

See *z/OS Language Environment Programming Reference* for more information about the CEE3SPM callable service.
**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 46 on page 250 is signaled, Language Environment builds the q_data structure shown in Figure 81 on page 251.

<table>
<thead>
<tr>
<th>Program interruption</th>
<th>Program interruption code</th>
<th>Condition</th>
<th>Message number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square-root exception</td>
<td>1D</td>
<td>CEE1UQ</td>
<td>2010</td>
</tr>
</tbody>
</table>

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.

**q_data structure for math and bit-manipulation conditions**

For conditions that occur in the mathematical or bit manipulation routines, the Language Environment condition manager creates q_data that user condition handlers can use to handle the condition. The q_data structure is shown in Figure 81 on page 251, and is the same for all entry points of the mathematical and bit manipulation routines.
The following information is provided by the q_data structure shown in Figure 81 on page 251:

**q_data_token (input)**
The 4-byte address of the address list. This value is returned by the CEEGQDT callable service.

**parm_count (input)**
A 4-byte binary integer containing the value 10, which is the total number of q_data fields in the q_data structure, including parm_count.

**math_operation (input)**
An 8-byte field containing an abbreviation for the mathematical or bit manipulation operation for which the condition occurred. The field is left-justified and padded with blanks. (See Table 47 on page 253 for a list of the abbreviations.)
The q_data descriptor for std_resume_value.

A default value used as the result of the mathematical or bit manipulation function when the user condition handler requests the resume action (result code 10). The length and type of this field are dependent on math_operation and are reflected in the q_data descriptor std_resume_value_desc.

The q_data descriptor for parameter_1.

The value of the first parameter provided to the mathematical or bit manipulation routine. The length and type of this field are dependent on math_operation and are reflected in the q_data descriptor parameter_1_desc.

This is the value of the first parameter that is used as input to the routine when the user condition handler requests a resume with new input value (result code 60 with a new condition token of CEE0CE).

The q_data descriptor for parameter_2 if the mathematical or bit manipulation routine has two input parameters. (If the routine has only one parameter, the q_data structure has an address slot for this field, but the address is not meaningful and the field must not be referenced.)

The value of the second parameter provided to the mathematical or bit manipulation routine if the routine has two input parameters. (If the routine has only one parameter, the q_data structure has an address slot for this field, but the address is not meaningful and the field must not be referenced.) The length and type of this field are dependent on math_operation and are reflected in the q_data descriptor parameter_2_desc.

This is the value of the second parameter that is used as input to the routine when the user condition handler requests a resume with new input value (result code 60 with a new condition token of CEE0CE).

The q_data descriptor for fixup_resume_value. (See “Format of q_data descriptors” on page 254 for more information about q_data descriptors.)

The value to be used as the result of the mathematical or bit manipulation function when the user condition handler requests a resume with new output value (result code 60 with a new condition token of CEE0CF). The length and type of this field are dependent on math_operation and are reflected in the q_data descriptor fixup_resume_value_desc.

The q_data descriptor for parameter_3 if the mathematical or bit manipulation routine has three input parameters. (If the routine has only one (or two) parameter(s), the q_data structure has an address slot for this field, but the address is not meaningful and the field must not be referenced.)

The value of the third parameter provided to the mathematical or bit manipulation routine if the routine has three input parameters. (If the routine has only one (or two) parameter(s), the q_data structure has an address slot for this field, but the address is not meaningful and the field must not be referenced.) The length and type of the field are dependent on math_operation and are reflected in the q_data descriptor parameter_3_desc.

This is the value of the third parameter that is used as input to the routine when the user condition handler requests a resume with new input value (result code 60 with a new condition token of CEE0CE).
Table 47: Abbreviations of math operations in q_data structures. Column two shows the abbreviations that can occur in field math_operation for the math operations shown in column one.

<table>
<thead>
<tr>
<th>Mathematical operation</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logarithm Base e</td>
<td>LN</td>
</tr>
<tr>
<td>Logarithm Base 10</td>
<td>LOG</td>
</tr>
<tr>
<td>Logarithm Base 2</td>
<td>LOG2</td>
</tr>
<tr>
<td>Exponential (base e)</td>
<td>E**Y</td>
</tr>
<tr>
<td>Exponentiation (x raised to the power y)</td>
<td>X**Y</td>
</tr>
<tr>
<td>Arcsine</td>
<td>ARCSIN</td>
</tr>
<tr>
<td>Arccosine</td>
<td>ARCCOS</td>
</tr>
<tr>
<td>Arctangent</td>
<td>ARCTAN</td>
</tr>
<tr>
<td>Arctangent2</td>
<td>ARCTAN2</td>
</tr>
<tr>
<td>Sine</td>
<td>SIN</td>
</tr>
<tr>
<td>Cosine</td>
<td>COS</td>
</tr>
<tr>
<td>Tangent</td>
<td>TAN</td>
</tr>
<tr>
<td>Cotangent</td>
<td>COTAN</td>
</tr>
<tr>
<td>Hyperbolic Sine</td>
<td>SINH</td>
</tr>
<tr>
<td>Hyperbolic Cosine</td>
<td>COSH</td>
</tr>
<tr>
<td>Hyperbolic Tangent</td>
<td>TANH</td>
</tr>
<tr>
<td>Hyperbolic Arctangent</td>
<td>ARCTANH</td>
</tr>
<tr>
<td>Square Root</td>
<td>SQRT</td>
</tr>
<tr>
<td>Error Function</td>
<td>ERF</td>
</tr>
<tr>
<td>Error Function Complement</td>
<td>ERFC</td>
</tr>
<tr>
<td>Gamma Function</td>
<td>GAMMA</td>
</tr>
<tr>
<td>Log Gamma Function</td>
<td>LOGGAMMA</td>
</tr>
<tr>
<td>Absolute Value Function</td>
<td>ABS</td>
</tr>
<tr>
<td>Modular Arithmetic</td>
<td>MOD</td>
</tr>
<tr>
<td>Truncation</td>
<td>TRUNC</td>
</tr>
<tr>
<td>Imaginary Part of Complex</td>
<td>IPART</td>
</tr>
<tr>
<td>Conjugate of Complex</td>
<td>CPART</td>
</tr>
<tr>
<td>Nearest Whole Number</td>
<td>NWN</td>
</tr>
<tr>
<td>Nearest Integer</td>
<td>NINT</td>
</tr>
<tr>
<td>Positive Difference</td>
<td>POSDIFF</td>
</tr>
<tr>
<td>Transfer of Sign</td>
<td>XFERSIGN</td>
</tr>
<tr>
<td>Floating Complex Multiply</td>
<td>CPLXMULT</td>
</tr>
<tr>
<td>Floating Complex Divide</td>
<td>CPLXDIVD</td>
</tr>
<tr>
<td>Bit Shift</td>
<td>ISHFT</td>
</tr>
</tbody>
</table>
**Usage notes**

- You can use the CEEGQDT callable service to retrieve the q_data_token; see *z/OS Language Environment Programming Reference* for details.
- From a Fortran routine, you can retrieve the qualifying data using Fortran-specific callable services and functions, which are described in *Language Environment for MVS & VM Fortran Run-Time Migration Guide*.
- A user condition handler can request one of three different actions to continue the execution of a failing mathematical or bit manipulation routine:
  - The resume action (result code 10). The value in std_resume_value (either the default value provided to the user condition handler or a modified value provided by the user condition handler) becomes the final result value for the routine.
  - The resume with new input value action (result code 60 with a new condition token of CEE0CE). The values to be used as parameters for invoking the routine again are provided by the user condition handler in parameter_1 and, if applicable, in parameter_2.
  - The resume with new output value action (result code 60 with a new condition token of CEE0CF). The fixup_resume_value value provided by the user condition handler becomes the final result value for the routine.

**Format of q_data descriptors**

q_data descriptors contain additional information you need to fix up the parameter or result fields of the math q_data structures, the result field of the program interruption q_data structures, or fields for any conditions whose q_data structures contain q_data descriptors. The descriptors contain information about the length and data type of these fields. The format of the q_data descriptor is illustrated in Figure 82 on page 254.

```
+0  X'02'   | data_type_1 | X'CE' | data_type_2 |
+4  length                                   
```

*Figure 82: Format of a q_data descriptor*

The following information is provided by the q_data descriptor shown in Figure 82 on page 254:

- **data_type_1**
  - A 1-byte binary integer value that, along with data_type_2, indicates the data type. See Table 48 on page 255 for the values and their corresponding data types.

- **data_type_2**
  - A 1-byte binary integer value that, along with data_type_1, indicates the data type. See Table 48 on page 255 for the values and their corresponding data types.

- **length**
  - A 4-byte binary integer value that represents the length of the data.
For each type code that can occur in a q_data descriptor, Table 48 on page 255 shows the corresponding data type.

Table 48: q_data descriptor data types

<table>
<thead>
<tr>
<th>data_type_1 type</th>
<th>data_type_2 type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>String of single-byte characters with no length prefix or ending delimiter</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>Signed binary integer whose length is 1, 2, 4, or 8 bytes</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>Floating-point number whose length is 4, 8, or 16 bytes</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>Complex number whose length is 8, 16, or 32 bytes</td>
</tr>
<tr>
<td>1</td>
<td>18</td>
<td>Unsigned binary integer whose length is 1 byte</td>
</tr>
</tbody>
</table>
Using condition tokens
Chapter 19. Using and handling messages

This topic describes how you can use the Language Environment message services to create, issue, and handle messages for Language Environment-conforming applications.

How Language Environment messages are handled

The Language Environment message services provide a common method of handling and issuing messages for Language Environment-conforming applications.

When a condition is raised in your application, either Language Environment common routines or language-specific runtime routines can issue messages from the runtime message file. The messages can provide information about the condition and suggest possible solutions to errors.

You can use Language Environment callable services and runtime options to modify message handling, and control the destination of message output. You can also define a message log file to create a record of the messages that Language Environment issues.

Related runtime options:

**MSGFILE**
- Specifies a file where runtime messages issued by Language Environment are logged.

**MSGQ**
- Specifies the maximum number of ISIs

**NATLANG**
- Specifies the national language runtime message file.

Related callable services:

**CEEMGET**
- Gets a message.

**CEEMOUT**
- Dispatches a message.

**CEEMSG**
- Gets, formats, and dispatches a message.

**CEECMI**
- Stores and loads message insert data about a condition.

Related utilities:

**CEEBLDXT**
- Transforms source files into loadable TEXT files.

See z/OS Language Environment Programming Reference for more information about the callable services syntax.

Creating messages

The following topics 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 CEEBLDXT 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 CEEBLDXT 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 83 on page 258 shows an example of a message source file with a facility ID of XMP.

```
:facid.XMP
:msgno.10
:msgsuid.0001
:msgname.EXMPLMSG
:msgclass.I
:msg.This is an example of an insert,
:tab.+1
:ins 1.a simple insert
:msg., within a message.
:msg.This is a simple example of how to put an insert into a message.
:sysact.No system action is taken.
```

Figure 83: 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 three 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 “Runtime messages with POSIX” on page 271.

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 :msgsuid. 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 "Runtime messages with POSIX" on page 271.

: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 CEEBLDXT 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 49 on page 270 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 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.
The `:tab.n` 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.

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.

The `:etbn` tag turns off the tabs set by a `:tbn` tag.

The `:ins n.[text]` tag defines a message insert. The insert is a variable that is assigned a value with the CEECMI 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.

The `:newline` tag creates a new message line that can be used for multiline messages.

The `:xpl` 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.

The `:presp` 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.

The `:sysact` 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**

The syntax is as follows:

```
ceedtfx   [-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:** The `ceedtfx` 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 that contains 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 Tout_file.

- **-I secondary_file_name**
  The name of the secondary input file that is generated for the language that is specified with the -l (lowercase L) option. If no suffix is present in the specified secondary_file_name, the extension is .h for C, .fortran for Fortran, and .copy for all others.

- **-P**
  This option is used to save previous prologs, if files being generated 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 specifies which COBOL delimiter to use. It is used in combination with the -l (lowercase L) COBOL option. By default, APOST is used as the delimiter.

  Escape quotation marks to prevent them from being treated as shell meta characters.

  Examples:

  ```
  ceelldtx -l COBOL -I secondary_file_name -d \' in_file out_file
  ceelldtx -l COBOL -I secondary_file_name -d \" in_file out_file
  ceelldtx -l COBOL -I secondary_file_name -d QUOTE in_file out_file
  ```

- **-l BAL | C | COBOL | FORTRAN | PLI**
  This option specifies the language to be used in generating a secondary input file. It is used in combination with the -l (uppercase l) secondary_file_name option. The file will contain declarations for the condition tokens that are associated with each message in the message source file. The language is accepted in lowercase and uppercase.

  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.
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.

**PRL | NOPRL**
Specify PRL to reuse prolog from the previous file version, if previous version exists. Specify NOPRL to ignore the previous prolog. PRL 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 267 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 83 on page 258 , issue the environment corresponding command example below. After the out_file is 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:**
ceedltxtx -l PLI -I exemplcop 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 these messages for CEEBLDTX errors.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
<th>Explanation</th>
<th>Programmer response</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>IRX0005I Machine storage</td>
<td>Machine storage exhausted</td>
<td>Attempt one of the following options:</td>
</tr>
<tr>
<td></td>
<td>exhausted</td>
<td></td>
<td>1. Increase the virtual storage space available on the system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Split up the script in_file, into two or more files. Adjust the message module table for the corresponding split.</td>
</tr>
<tr>
<td>0005</td>
<td>Error reading file sssssss.</td>
<td>Error occurred while reading file sssssss.</td>
<td>Validate file accessibility.</td>
</tr>
<tr>
<td>0006</td>
<td>Error erasing file sssssss.</td>
<td>Error occurred while erasing file sssssss.</td>
<td></td>
</tr>
</tbody>
</table>
Validate file accessibility.

**Return Code=0007**

**Explanation**
Error occurred while writing file sssssss.

**Programmer response**
Make sure the CSECT name is 63 characters or less.

**Return Code=0008**

**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**

**Explanation**
Option specified must be accompanied by an argument.

**Programmer response**
Specify an argument with the option.

**Return Code=0010**

**Explanation**
Invalid option specified.

**Programmer response**
Specify zero or more valid options.

**Return Code=0011**

**Explanation**
The data set name is not correctly specified.

**Programmer response**
Validate the name of the data set is correct.

**Return Code=0020**

**Explanation**
The CSECT name sssssss 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**

**Explanation**
The CSECT name sssssss does not begin with a letter, $, #, @ or underscore (_).

**Programmer response**
Make sure that the CSECT name sssssss begins with a letter, $, #, @ or underscore (_).

**Return Code=0028**

**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**

**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**

**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: IWESCFA.

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.

**Return**

**Code=0088**

**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**

**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**

**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**

**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**

**Explanation**
Insert number was not provided or was less than 1 on line nnn.

**Programmer response**
Specify a positive insert number of 9 or less for the insert.

**Return**

**Code=0104**

**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**

**Explanation**
A secondary file with the name sssssss does not exist on A-disk.

**Programmer response**
Make sure the name is given correctly and is accessible.

**Return**

**Code=0112**

**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**

**Explanation**
An undefined error was encountered.
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 84 on page 267, 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
DC F'03'              lowest message number in module
DC F'100'             highest message number in module
DC CL8'EXPLASM'       message module name
DC F'-1'                16-byte entry (a dummy entry)...
DC CL8'DUMMY'          in the language table
END  UXMPMSGT
```

Figure 84: Example of a message module table with one language

In the message module table in Figure 85 on page 267, 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.

```
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
DC F'03'              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'                16-byte entry (a dummy entry)...
DC CL8'DUMMY'          in the language table
END  UZOGMSGT
```

Figure 85: 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
Assigning values to message inserts

 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 85 on page 267, 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 84 on page 267; the title of the message module table for messages having a facility ID of ZOG would be ‘UZOGMSGT’ as shown in Figure 85 on page 267.

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 callable service CEECMI 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. Before invoking the CEECMI callable service, assign values to the callable service parameters. For more information about CEECMI, see z/OS Language Environment Programming Reference.

Figure 86 on page 269 shows an example of the use of CEECMI to assign value 1234 to insert 1 for :msgname.EXMPLMSG shown in Figure 83 on page 258.
*PROCESS MACRO;
TEST: Proc Options(Main);
/*Module/File Name: IBMMINS  */
%INCLUDE  CEEIBMAW;
%INCLUDE  CEEIBMCT;
%INCLUDE SYSLIB(EXMPLCOP);
DECLARE INSERT    CHAR(255) VARYING  AUTO;
DCL 01 CTOK,                    /* Feedback token */
     03 MsgSev    REAL FIXED BINARY(15,0),
     03 MsgNo     REAL FIXED BINARY(15,0),
     03 Flags,    /* Flags */
     05 Case      BIT(2),
     05 Severity  BIT(3),
     05 Control   BIT(3),
     03 FacID     CHAR(3),    /* Facility ID */
     03 ISI                   /* Instance-Specific Information */
     REAL FIXED BINARY(31,0);
DCL 01 FBCODE,                  /* Feedback token */
     03 MsgSev    REAL FIXED BINARY(15,0),
     03 MsgNo     REAL FIXED BINARY(15,0),
     03 Flags,    /* Flags */
     05 Case      BIT(2),
     05 Severity  BIT(3),
     05 Control   BIT(3),
     03 FacID     CHAR(3),    /* Facility ID */
     03 ISI                   /* Instance-Specific Information */
     REAL FIXED BINARY(31,0);
DECLARE MSGFILE   FIXED AUTO;
ctok = EXMPLMSG;
insert = ’1234’;
MSGFILE = 2;
/* Call CEECMI to create a message insert */
CALL CEECMI(ctok, 1, insert, fbcode);
/* Call CEEMSG to issue the message */
CALL CEEMSG(ctok, MSGFILE, fbcode);
END TEST;

Figure 86: Example of assigning values to message inserts

Interpreting runtime 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 runtime routines issue runtime messages. All runtime messages in Language Environment are composed of the following:

- A 3-character facility ID used by all messages that are 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 that is associated with the condition to MSGFILE. For more information about the condition token, see Chapter 18, “Using condition tokens,” on page 233.
- A message number that identifies the message that is associated with the condition.
- A severity level that indicates the severity of the condition that was raised.

The format of every runtime message is FFFnnnnx
**Interpreting runtime messages**

**FFF**
- Represents the facility ID. In z/OS Language Environment, the possible facility IDs assigned by IBM are:
  - **CEE** Language Environment common library.
  - **EDC** C language-specific library.
  - **FOR** Fortran language-specific library.
  - **IGZ** COBOL language-specific library.
  - **IBM** PL/I language-specific library

**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 49 on page 270 lists the severity codes, corresponding severity levels, explanations of the severity codes, and the default actions that are taken if conditions corresponding to each level of severity are unhandled.

<table>
<thead>
<tr>
<th>Severity code</th>
<th>Level of severity</th>
<th>Explanation</th>
<th>Default action if condition unhandled</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0</td>
<td>An informational message (or, if the entire token is zero, no information).</td>
<td>No message issued.</td>
</tr>
<tr>
<td>W</td>
<td>1</td>
<td>A warning message; service completed, probably successfully.</td>
<td>No message issued, except in COBOL. Processing continues for all languages.</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>Error detected, correction attempted, service completed, perhaps successfully.</td>
<td>Issues message and terminates thread.</td>
</tr>
<tr>
<td>S</td>
<td>3</td>
<td>Severe error detected, service incomplete with possible side effects.</td>
<td>Issues message and terminates thread.</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>Critical error detected, service incomplete with condition signaled.</td>
<td>Issues message and terminates thread.</td>
</tr>
</tbody>
</table>

Language Environment messages can appear even though you made no explicit calls to Language Environment services. C, COBOL, and PL/I runtime library routines commonly use the Language Environment services, so you might receive Language Environment messages even when the application routine does not directly call Language Environment services.

Some Language Environment conditions have qualifying data that is associated with the instance-specific information (ISI) for the condition. For more information about qualifying data, see “q_data structure for abends” on page 244.
Specifying national languages

You can use Language Environment national language support to view runtime 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.

The CEE3LNG callable service allows you to set or query the current national language setting while your routine is running. Refer to z/OS Language Environment Programming Reference for more information about CEE3LNG.

Runtime messages with POSIX

The POSIX(ON) runtime option changes both the facility ID and message number for some messages you might see with your C application. Messages that had the facility ID of EDC and ranged in number from 6000 through 6009 before running POSIX(ON) now have a facility ID of CEE and different message numbers.

Table 50 on page 271 shows the conditions, their facility ID and message number for the different runtime environments. If your C application is coded to respond to specific facility IDs or specific message numbers for processing, then you must check for the proper values depending on the environment.

<table>
<thead>
<tr>
<th>Condition token</th>
<th>Facility ID with POSIX(ON)</th>
<th>Message number with POSIX(ON)</th>
<th>Facility ID with POSIX(OFF)</th>
<th>Message number with POSIX(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGFPE</td>
<td>CEE</td>
<td>5201</td>
<td>EDC</td>
<td>6000</td>
</tr>
<tr>
<td>SIGILL</td>
<td>CEE</td>
<td>5202</td>
<td>EDC</td>
<td>6001</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>CEE</td>
<td>5203</td>
<td>EDC</td>
<td>6002</td>
</tr>
<tr>
<td>SIGABND</td>
<td>CEE</td>
<td>5204</td>
<td>EDC</td>
<td>6003</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>CEE</td>
<td>5205</td>
<td>EDC</td>
<td>6004</td>
</tr>
<tr>
<td>SINGINT</td>
<td>CEE</td>
<td>5206</td>
<td>EDC</td>
<td>6005</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>CEE</td>
<td>5207</td>
<td>EDC</td>
<td>6006</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>CEE</td>
<td>5208</td>
<td>EDC</td>
<td>6007</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>CEE</td>
<td>5209</td>
<td>EDC</td>
<td>6008</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>CEE</td>
<td>5210</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>CEE</td>
<td>5211</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>CEE</td>
<td>5212</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>CEE</td>
<td>5213</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>CEE</td>
<td>5214</td>
<td>na</td>
<td>na</td>
</tr>
</tbody>
</table>
Table 50: Condition tokens with POSIX (continued)

<table>
<thead>
<tr>
<th>Condition token</th>
<th>Facility ID with POSIX(ON)</th>
<th>Message number with POSIX(ON)</th>
<th>Facility ID with POSIX(OFF)</th>
<th>Message number with POSIX(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGCONT</td>
<td>CEE</td>
<td>5215</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>CEE</td>
<td>5216</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGTIN</td>
<td>CEE</td>
<td>5217</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>CEE</td>
<td>5218</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGIO</td>
<td>CEE</td>
<td>5219</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>CEE</td>
<td>5220</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>CEE</td>
<td>5221</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>CEE</td>
<td>5222</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGIOERR</td>
<td>CEE</td>
<td>5223</td>
<td>EDC</td>
<td>6009</td>
</tr>
<tr>
<td>SIGDCE</td>
<td>CEE</td>
<td>5224</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGPOLL</td>
<td>CEE</td>
<td>5225</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGURG</td>
<td>CEE</td>
<td>5226</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>CEE</td>
<td>5227</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>CEE</td>
<td>5228</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGWINCH</td>
<td>CEE</td>
<td>5229</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGXCPU</td>
<td>CEE</td>
<td>5230</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGXFSZ</td>
<td>CEE</td>
<td>5231</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGVTALRM</td>
<td>CEE</td>
<td>5232</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>CEE</td>
<td>5233</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGDUMP</td>
<td>CEE</td>
<td>5234</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGNEDANGER</td>
<td>CEE</td>
<td>5235</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGTHSTOP</td>
<td>CEE</td>
<td>5236</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>SIGTHCONT</td>
<td>CEE</td>
<td>5237</td>
<td>na</td>
<td>na</td>
</tr>
</tbody>
</table>

Handling message output

The following topics provide information about directing message output and displaying messages under Language Environment, C, C++, COBOL, Fortran, and PL/I.

For information about handling message output in ILC applications, see z/OS Language Environment Writing Interlanguage Communication Applications.

Using Language Environment MSGFILE

Runtime messages are directed to a common Language Environment message file. You can use the MSGFILE runtime option to specify the ddname of this file. If a message file ddname is not declared, messages are written to the IBM-supplied default ddname SYSOUT.

The definitions of MSGFILE(SYSOUT) differ, depending on the operating system you use. Table 51 on page 273 lists the SYSOUT definitions and MSGFILE default attributes for MVS and TSO/E:
Table 51: Operating system, SYSOUT definitions, MSGFILE default attributes

<table>
<thead>
<tr>
<th>Operating system</th>
<th>SYSOUT definition</th>
<th>MSGFILE default attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVS</td>
<td>SYSOUT=*</td>
<td>LRECL 121, RECFM FBA</td>
</tr>
<tr>
<td></td>
<td>The output is routed to the destination specified in the MSGCLASS option of the JOB card.</td>
<td>If not a terminal, BLKSIZE 121*100; if a terminal, BLKSIZE 121.</td>
</tr>
<tr>
<td>TSO/E</td>
<td>ALLOC DD(SYSOUT) DA(*)</td>
<td>LRECL 121, RECFM FBA, BLKSIZE 121</td>
</tr>
</tbody>
</table>

When you direct runtime messages to an I/O device, the method you should use also depends on the operating system. Table 52 on page 273 lists methods for directing runtime messages to an I/O device under MVS and TSO/E, and provides references for additional information about this topic.

Table 52: Defining an I/O device for a ddname

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Method to define I/O device</th>
<th>For more information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVS</td>
<td>Specify the ddname of a data set in the JCL.</td>
<td>&quot;Required DD Statements&quot; in “Writing JCL for the link-edit process” on page 60</td>
</tr>
<tr>
<td>TSO/E</td>
<td>The ddname of the data set that you specify using the ALLOCATE command.</td>
<td>Chapter 6, “Creating and executing programs under TSO/E,” on page 71</td>
</tr>
</tbody>
</table>

Note:

1. You need to modify existing JCL of pre-Language Environment-conforming applications in order to define new ddnames for MSGFILE.
2. You can specify the same message file across nested enclaves. Language Environment coordinates the use of the same ddname across nested enclaves. If you specify different MSGFILE ddnames in each enclave, Language Environment honors each ddname.
3. The Language Environment MSGFILE can be allocated to a large format sequential data set.
4. Under CICS, the MSGFILE runtime option is ignored. All runtime messages are directed to a transient data queue named CESE rather than to the ddname specified in the MSGFILE option. For more information about message handling and runtime message output under CICS, see “Runtime output under CICS” on page 362.

Using MSGFILE under z/OS UNIX

To direct MSGFILE output to a z/OS UNIX file, use the PATH= keyword in the ddname parameter of MSGFILE to specify a ddname that nominates a z/OS UNIX file.

If your application is running in an address space created by using the fork() or spawn() functions or if it is invoked by one of the exec family of functions, the application has access to a DD card only if you dynamically allocate one. If the application can access a DD card, MSGFILE output is directed to that file. If the allocated DD card contains the PATH= keyword, Language Environment directs the MSGFILE output to the specified file in the z/OS UNIX file system.

If your application is running under z/OS UNIX, or under any environment that has file descriptor 2 (FD2) open, MSGFILE output is directed to whatever FD2 points to. Under the shell this is typically your terminal.

If FD2 does not exist but your application is either running in an address space created by the fork() or spawn() functions or invoked by one of the exec family of functions, MSGFILE output is directed to the current working directory; if that directory is the root directory, the output is written to a file in the
directory /tmp. The name of the file is the name you specify with the MSGFILE runtime option, with the default of SYSOUT.

The resulting file name has the following format:

```
/path/Fname.Date.Time.Pid
```

**path**

The current working directory (unless it is the working directory, in which case it is then /tmp).

**Fname**

The name specified in the FNAME parameter on the call to CEE3DMP (default is CEEDUMP).

**Date**

The date the dump is taken, appearing in the format YYYYMMDD (such as 19940325 for March 25, 1994).

**Time**

The time the dump is taken, appearing in the format HHMMSS (such as 175501 for 05:55:01 PM).

**Pid**

The process ID the application is running in when the dump is taken.

**Note:** Language Environment cannot direct MSGFILE output to a z/OS UNIX file in a CICS environment.

### Using C or C++ I/O functions

C and C++ make a distinction between types of error output, and whether the output is directed to the MSGFILE destination or to one of the standard stream output devices, stderr or stdout.

Runtime messages and perror() messages are directed to the stderr standard stream output device. The default destination for stderr output is the MSGFILE ddname; you can change this default as discussed below.

Message output issued by a call to the printf() function is directed to stdout. For TSO/E, stdout defaults to the terminal. When running batch (MVS, IMS, or TSO/E) or IMS online, stdout attempts by default to open one of several ddnames in the following order of precedence, which is made to open SYSOUT=* as a data set:

1. SYSPRINT
2. SYSTERM
3. SYSERR

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 53 on page 274 lists the types of C/C++ output, the types of messages associated with them, and the destination of the message output.

<table>
<thead>
<tr>
<th>Type of output</th>
<th>Type of message</th>
<th>Produced by</th>
<th>Default destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSGFILE output</td>
<td>Language Environment</td>
<td>Language Environment</td>
<td>MSGFILE ddname</td>
</tr>
<tr>
<td></td>
<td>messages (CEExxxx)</td>
<td>unhandled conditions</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>C library messages</strong></td>
<td><strong>C/C++ unhandled</strong></td>
<td>MSGFILE ddname</td>
</tr>
<tr>
<td></td>
<td>conditions (EDCxxxx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stderr messages</td>
<td>perror() messages</td>
<td>Issued by a call to</td>
<td>MSGFILE ddname</td>
</tr>
<tr>
<td></td>
<td>(EDCxxx)</td>
<td>perror()</td>
<td></td>
</tr>
<tr>
<td>User output sent</td>
<td>explicitly to stderr</td>
<td>Issued by a call to</td>
<td>MSGFILE ddname</td>
</tr>
<tr>
<td></td>
<td>issued to stdout</td>
<td>fprintf()</td>
<td></td>
</tr>
</tbody>
</table>

Table 53: C and C++ message output.
Table 53: C and C++ message output. (continued)

<table>
<thead>
<tr>
<th>Type of output</th>
<th>Type of message</th>
<th>Produced by</th>
<th>Default destination</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>stdout messages</strong></td>
<td>User output sent explicitly to stdout</td>
<td>Issued by a call to printf()</td>
<td>stdout</td>
</tr>
</tbody>
</table>

You can control the destination of stderr and stdout output by using the Language Environment MSGFILE runtime option, the C freopen() function, or by invoking redirection services at run time.

Table 54 on page 275 lists the possible destinations of redirected stderr and stdout standard stream output.

Table 54: C/C++ redirected stream output.

<table>
<thead>
<tr>
<th>Action</th>
<th>stderr not redirected</th>
<th>stderr redirected to destination other than stdout</th>
<th>stderr redirected to stdout</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdout not redirected</td>
<td>stdout to itself</td>
<td>stdout to itself</td>
<td>Both to stdout</td>
</tr>
<tr>
<td><strong>stderr to MSGFILE</strong></td>
<td></td>
<td>stderr to its other destination</td>
<td>n/a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stdout redirected to destination other than stderr</th>
<th>stdout to its other destination</th>
<th>stderr redirected to stdout</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>stderr to MSGFILE</strong></td>
<td>stderr to its other destination</td>
<td>n/a</td>
</tr>
</tbody>
</table>

| stdout redirected to stderr | Both to MSGFILE | 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 COBOL I/O statements

Language Environment manages all COBOL output directed to the system-logical output device. This includes output from:

- DISPLAY SYSOUT
- READY TRACE (OS/VS COBOL only)
- EXHIBIT (OS/VS COBOL only)

Note: For OS/VS COBOL programs running under CICS, the DISPLAY, READY TRACE and EXHIBIT statements are not supported.

Non-CICS considerations

For COBOL programs, the DISPLAY statement sends output to MSGFILE(SYSOUT), the default ddname for the Language Environment message file. You can use the COBOL OUTDD compiler option to change the destination of DISPLAY output. The MVS data set to which the runtime messages are written depends on the combination of ddnames specified in the OUTDD compiler option and the MSGFILE runtime option.
Handling message output

If the ddname in OUTDD matches the ddname specified in the MSGFILE runtime option, the output is synchronized with the runtime messages and placed in the MVS data set designated by the MSGFILE runtime option.

If the ddname in OUTDD does not match the ddname specified in the MSGFILE runtime option, the output from the DISPLAY statement is directed to the OUTDD ddname destination.

If the file designated by MSGFILE has not been defined (associated with an I/O device) when the output is delivered, Language Environment dynamically allocates the file with ddname and attributes as shown in Table 51 on page 273.

If the file designated by OUTDD has not been defined when the output is delivered, Language Environment dynamically allocates the file with ddname and attributes as shown in Table 51 on page 273.

For more information about directing COBOL output, see the appropriate version of the COBOL programming guide in the COBOL library at Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733).

CICS considerations

DISPLAY to the system-logical output device is supported under CICS for programs compiled with VS COBOL II, COBOL for MVS & VM, COBOL for OS/390 & VM, and Enterprise COBOL for z/OS. The DISPLAY output is written to the Language Environment message file (transient data queue CESE).

Using Fortran I/O statements

Under Language Environment, Fortran I/O statements formerly written to a Fortran error message unit (either directly or by default) are directed to a Language Environment message file specified by the MSGFILE runtime option. At program initialization, the Fortran error message unit is connected to the file specified by the MSGFILE runtime option.

The following types of output from Fortran programs are directed to the message file:

- Error messages that result from unhandled conditions
- Output produced by a sequential WRITE statement with a unit identifier having a value equal to the Fortran error message unit
- Output produced by a sequential WRITE statement with * as the unit identifier when the Fortran error message and the standard print unit are the same
- Output produced by a PRINT statement when the Fortran error message and the standard print unit are the same
- Printed output from the dump services (CDUMP, CPCUMP, DUMP, PDUMP, or SDUMP)

The message file can be either a named or unnamed file. To specify an unnamed file, code the MSGFILE runtime option as follows:

```plaintext
MSGFILE(FTeeF001)
```

The ee value is a two-character representation of the error message unit number that is specified in the ERRUNIT runtime option; the ee value cannot be any other number.

The default ddname of the Language Environment message file is SYSOUT. The ddname can be changed in a Fortran program by issuing an OPEN statement to connect to the error message unit with a different ddname in the FILE specifier. You can use a CLOSE statement to close the message file currently connected to the Fortran error message unit. In this case, the default message file as specified by the MSGFILE runtime option becomes the current message file. Any subsequent output messages are written to this message file after the CLOSE statement is issued. No subsequent OPEN statement is required.

For example, when the standard print unit is the same as the error message unit (unit 6 in Figure 87 on page 277), all output from the PRINT statement is directed to the error message unit. When the MSGFILE(ONE) runtime option is in effect, the message file with ddname ONE is connected to the error message unit.
PRINT *, 'FILE ONE, RECORD 1'
PRINT *, 'FILE ONE, RECORD 2'
OPEN (6, FILE='TWO')
PRINT *, 'FILE TWO, RECORD 1'
PRINT *, 'FILE TWO, RECORD 2'
CLOSE (6)
PRINT *, 'FILE ONE, NEW RECORD 1'
PRINT *, 'FILE ONE, NEW RECORD 2'

Figure 87: Directing output messages

Figure 87 on page 277 shows the first two records being written to the message file with the ddname ONE. The first OPEN statement closes file ONE and connects file TWO to the error message unit; two messages are written to it. The CLOSE statement closes file TWO and makes ONE the current message file. This occurs because the MSGFILE(ONE) runtime option is specified. The next PRINT statement connects file ONE to the error message unit, and two records are written to it. The message file is opened and the error message unit is connected automatically when an output message is issued.

The error message unit is restricted to sequential formatted output operations. Therefore, there are restrictions on the OPEN statement specifiers that can be used for the error message unit. Table 55 on page 277 shows the valid OPEN statement specifiers and specifier values.

<table>
<thead>
<tr>
<th>SPECIFIER=spv</th>
<th>Default spv value</th>
<th>Additional allowable spv values</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS=sta</td>
<td>UNKNOWN</td>
<td>None</td>
</tr>
<tr>
<td>ACCESS=acc</td>
<td>SEQUENTIAL</td>
<td>None</td>
</tr>
<tr>
<td>CHAR=chr</td>
<td>DBCS</td>
<td>NODBCS can also be specified, but is ignored.</td>
</tr>
<tr>
<td>FORM=frm</td>
<td>FORMATTED</td>
<td>None</td>
</tr>
<tr>
<td>ACTION=act</td>
<td>WRITE</td>
<td>None</td>
</tr>
<tr>
<td>BLANK=blk</td>
<td>ZERO</td>
<td>NULL. BLANK has no meaning because the error message unit is used only for output.</td>
</tr>
<tr>
<td>PAD=pad</td>
<td>YES</td>
<td>NO. PAD has no meaning because the error message unit is used only for output.</td>
</tr>
<tr>
<td>POSITION=ASIS</td>
<td>ASIS</td>
<td>None</td>
</tr>
<tr>
<td>DELIM=dlm</td>
<td>Based on Fortran OPEN and CLOSE statements that refer to the error message unit.</td>
<td>APOSTROPHE, QUOTE, or NONE can be specified.</td>
</tr>
<tr>
<td>RECL=rcl</td>
<td>Maximum data length of a message file record.</td>
<td>Any positive value which does not exceed the maximum allowable length of the data in a message file record.</td>
</tr>
</tbody>
</table>

Using PL/I I/O statements

Runtime messages in PL/I routines are directed to the file specified by the Language Environment MSGFILE runtime option, instead of to the PL/I SYSPRINT STREAM PRINT file.

User-specified output is still directed to the PL/I SYSPRINT STREAM PRINT file by default. To direct this output to the Language Environment MSGFILE file, specify the runtime option MSGFILE(SYSPRINT).
When you use MSGFILE(SYSPRINT):

- Any file constant declaration that includes SYSPRINT STREAM PRINT file attributes is ignored.
- File attributes specified in the SYSPRINT DD card are used.
- If SYSPRINT DD is not present at first file reference, Language Environment dynamically allocates a file with IBM-supplied attributes. See Table 51 on page 273 for MSGFILE file default attributes.
- Any OPENs and CLOSEs to the PL/I SYSPRINT STREAM PRINT file are ignored.
- Synchronization between the types of output (messages and user-specified output) is not provided, so the order of the output is unpredictable.

**MSGFILE considerations when using PL/I**

If MSGFILE(SYSPRINT) is in effect, use SYSPRINT only to direct output to the PL/I SYSPRINT STREAM PRINT file.

Because performance is slower with the MSGFILE(SYSPRINT) option, it is recommended only for debugging purposes. For production applications, direct user-created output to the PL/I SYSPRINT STREAM PRINT file.

In a nested enclave environment, you can specify MSGFILE(SYSPRINT) for all enclaves in the application or only for those enclaves containing PUT statements. For batch, multiple enclaves in a Language Environment process can use the PL/I SYSPRINT STREAM PRINT. In this instance, you cannot open the file until it is referenced, and it is closed by Language Environment at process termination.

Under CICS, the MSGFILE runtime option is ignored. Both runtime messages and the SYSPRINT STREAM PRINT file output are directed to the CESE transient data queue. The CESE transient data queue is a CICS thread-level resource. See Chapter 25, “Running applications under CICS,” on page 351 for more information about the CESE transient data queue.

For more information about directing PL/I output, refer to IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

**Examples using multiple message handling callable services**

The examples in this topic show how to use the Language Environment message and condition-handling services to issue a message that relates to a condition token. The same calls are illustrated in C/C++, PL/I, and COBOL.

Each example illustrates how CEEMOUT dispatches an informational message and uses CEENCOD to construct a token for the message. The message area is then initialized, CEEMGET retrieves the message, and CEEDCOD decodes the feedback token from CEEMGET. After all of the message has been retrieved, CEEMOUT issues the message. If any of the services fail, CEEMSG issues an informational error message.

**C/C++ example calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG**

```c
/*Module/File Name:  EDCMSGS */
/*****************************/
/**                      */
/**FUNCTION  :  CEEMOUT - dispatch a message to message file */
/**            :  CEENCOD - construct a condition token */
/**            :  CEEMGET - retrieve, format and store a message */
/**            :  CEEDCOD - decode an existing condition token */
/**            :  CEEMSG - retrieve, format, and dispatch a */
/**      */
/**              */
/**   This example illustrates the invocation of the Language Environment message and condition handling services. */
/**   It constructs a condition token, retrieves the associated message, and outputs the message to the message file. */
/**   This example program outputs the Language Environment message,"CEE0260S". */
```

278  z/OS: Language Environment Programming Guide
```c
#include <string.h>
#include <stdio.h>
#include <leawi.h>
#include <stdlib.h>
#include <ceeedcct.h>

int main(void) {
    _VSTRING message;
    _INT4 dest, msgindx;
    _CHAR80 msgarea;
    _FEEDBACK fc, token;
    _INT2 c_1, c_2, cond_case, sev, control;
    _CHAR3 facid;
    _INT4 isi;

    printf ( "\n**********************************\n" );
    printf ( "\nCE92MSG C Example is now in motion\n" );
    printf ( "\n**********************************\n" );
    strcpy(message.string,"The following message, CEE0260S, is expected");
    message.length = strlen(message.string);
    dest = 2;

    /*********************************************************************/
    /*   Call CEEMOUT to output informational message.               */
    /*   Call CEEMSG to output error message if CEEMOUT fails.      */
    /*********************************************************************/
    CEEMOUT(&message,&dest,&fc);
    if ( _FBCHECK (fc, CEE000) != 0 ) {
        /* put the message if CEEMOUT failed */
        dest = 2;
        CEEMSG(&fc,&dest,NULL);
        exit(2999);
    }

    /*********************************************************************/
    /* Construct a token for CEE message 0260.                       */
    /*********************************************************************/
    c_1 = 3;
    c_2 = 260;
    cond_case = 1;
    sev = 3;
    control = 1;
    memcpy(facid,"CEE",3);
    isi = 0;
    CEENCOD(&c_1,&c_2,&cond_case,&sev,&control,
            facid,&isi,&token,&fc);
    if ( _FBCHECK (fc, CEE000) != 0 ) {
        printf("CEENCOD failed with message number %d\n",
               fc.tok_msgno);
        exit(2999);
    }

    /*********************************************************************/
    /* Initialize the message area.                                  */
    /*********************************************************************/
    msgindx = 0;
    memset(msgarea,' ',79);
    msgarea[80] = '\0';

    /*********************************************************************/
    /* Use CEEMGET until all the message has been retrieved.        */
    /* Msgindx will be zero when all the message has been retrieved.*/
    /* Call CEEMSG to output error message if CEEMGET fails.        */
    /*********************************************************************/
    do {
        CEEMGET(&token,msgarea,&msgindx,&fc);
        if (fc.tok_sev > 1) {
            dest = 2;
            CEEMSG(&fc,&dest,NULL);
            exit(2999);
        }
        memcpy(message.string,msgarea,80);
        message.length = 80;
        dest = 2;
        CEEMOUT(&message,&dest,&fc); /* put out the message */
```
if ( _FBCHECK (fc , CEE000) != 0 ) {
    dest = 2;
    CEEMSG(&fc,&dest,NULL);
    exit(2999);
} while (msgindx != 0);
printf ( "\n
**********************************
" );
printf ( "{nCE92MSG C Example is now ended \n" );
printf ( "\n
**********************************
" );

COBOL example calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG

CBL LIB,QUOTE
*Module/File Name: IGZTMSGS
*******************************************************************************
*/* CE92MSG - Program to invoke the following LE services: *
*/* : CEEMOUT - dispatch a message to message file *
*/* : CEENCOD - construct a condition token *
*/* : CEEMGET - retrieve, format and store a message *
*/* : CEEDCOD - decode an existing condition token *
*/* : CEEMSG - retrieve, format, and dispatch a message to message file *
*/* This example illustrates the invocation of the Language Environment Message and Condition Handling services. *
*/* It constructs a condition token, retrieves the associated message, and outputs the message to the message file. *
*/* This example program will output the Language Environment message, "CEE0260S". *
********************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. CE92MSG.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MSGSTR.
   02 Vstring-length PIC S9(4) BINARY.
   02 Vstring-text.
   03 Vstring-char PIC X OCCURS 0 TO 256 TIMES DEPENDING ON Vstring-length of MSGSTR.
01 MSGDEST PIC S9(9) BINARY.
01 SEV PIC S9(4) BINARY.
01 MSGNO PIC S9(4) BINARY.
01 CASE PIC S9(4) BINARY.
01 SEV2 PIC S9(4) BINARY.
01 CNTRL PIC S9(4) BINARY.
01 FACID PIC X(3).
01 ISINFO PIC S9(9) BINARY.
01 MSGINDEX PIC S9(9) BINARY.
01 CTOK.
   02 Condition-Token-Value.
COPY CEEIGZCT.
   03 Case-1-Condition-ID.
     04 Severity PIC S9(4) BINARY.
     04 Msg-No PIC S9(4) BINARY.
   03 Case-2-Condition-ID
      REDEFINES Case-1-Condition-ID.
     04 Class-Code PIC S9(4) BINARY.
     04 Cause-Code PIC S9(4) BINARY.
   03 Case-Sev-Ctl PIC X.
   03 Facility-ID PIC XXX.
   02 I-S-Info PIC S9(9) BINARY.
01 FC.
   02 Condition-Token-Value.
COPY CEEIGZCT.
   03 Case-1-Condition-ID.
     04 Severity PIC S9(4) BINARY.
     04 Msg-No PIC S9(4) BINARY.
   03 Case-2-Condition-ID
      REDEFINES Case-1-Condition-ID.
     04 Class-Code PIC S9(4) BINARY.
     04 Cause-Code PIC S9(4) BINARY.

280 z/OS: Language Environment Programming Guide
PROCEDURE DIVISION.
 0001-BEGIN-PROCESSING.
  DISPLAY "**************************************".
  DISPLAY "CE92MSG COBOL Example is now in motion.".
  DISPLAY "**************************************".
  MOVE 80 TO Vstring-length of MSGSTR.
  MOVE "The following error message, CEE0260S, is expected:" TO Vstring-text of MSGSTR.
  MOVE 2 TO MSGDEST.
  ************************************************************
  **   Call CEEMOUT to put out informational message. **
  ************************************************************
  CALL "CEEMOUT" USING MSGSTR , MSGDEST , FC.
  IF NOT CEE000 of FC  THEN
      DISPLAY "Error " Msg-No of FC
      " in issuing header message"
      STOP RUN
  END-IF.
  ************************************************************
  **   Set up token fields for creation of a condition token  **
  ************************************************************
  MOVE   3 TO SEV.
  MOVE 260 TO MSGNO.
  MOVE   1 TO CASE.
  MOVE   3 TO SEV2.
  MOVE   1 TO CNTRL.
  MOVE "CEE" TO FACID.
  MOVE   0 TO ISINFO.
  ************************************************************
  **   Call CEENCOD to construct a condition token            **
  ************************************************************
  CALL "CEENCOD" USING SEV, MSGNO, CASE, SEV2, CNTRL, FACID, ISINFO, CTOK, FC.
  IF CEE000 of FC THEN
      MOVE 0 TO MSGINDX
      MOVE SPACES TO MSGAREA
  ************************************************************
  **       Call CEEMGET to retrieve message 260. Since        **
  **       message 260 is longer than the length of MSGAREA,  **
  **       a PERFORM statement loop is used to call CEEMGET   **
  **       multiple times until the message index is zero.   **
  ************************************************************
  PERFORM TEST AFTER UNTIL( MSGINDX = 0 )
    CALL "CEEMGET" USING CTOK, MSGAREA, MSGINDX, MGETFC
    IF (MGETFC NOT = LOW-VALUE) THEN
    ************************************************************
    **   Call CEEDCOD to decode CEEMGET's feedback token      **
    ************************************************************
    CALL "CEEDCOD" USING MGETFC, SEV, MSGNO, CASE, SEV2, CNTRL, FACID, ISINFO, FC
    IF NOT CEE000 of FC THEN
    ************************************************************
    **   Call CEEMSG to output LE error message               **
    **   using feedback code from CEEDCOD call.               **
    ************************************************************
    CALL "CEEMSG" USING MGETFC, MSGDEST, FC
    IF NOT CEE000 of FC THEN
    DISPLAY "Error " Msg-No of FC
    " from CEEMSG after error in CEEDCOD"
    END-IF
    STOP RUN
  END-IF.
  ************************************************************
  *                If decoded message number is not 455,      **
  *                then CEEMGET actually failed with error.   **
  ************************************************************
IF (Msg-No of MGETFC NOT = 455) THEN
DISPLAY "Error " Msg-No of MGETFC
" retrieving message CEE0260S"
STOP RUN
END-IF
END-IF
*********************************************
*       Call CEEMOUT to output each portion of message 260 **
*********************************************
MOVE MSGAREA TO Vstring-text of MSGSTR
CALL "CEEMOUT" USING MSGSTR, MSGDEST, FC
IF (MSGINDX = ZERO) THEN
DISPLAY "**************************************"
DISPLAY " COBOL message example program ended."
DISPLAY "**************************************"
END-IF
END-PERFORM
ELSE
DISPLAY "Error " Msg-No of FC
" in encoding condition token"
STOP RUN
END-IF.
GOBACK.

PL/I example calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG

*PROCESS MACRO;
/*Module/File Name: IBMMSGS */
CE92MSG: PROC OPTIONS(MAIN);
%INCLUDE  CEEIBMAW;
%INCLUDE  CEEIBMCT;
/*************************************************************/
/*                                                           */
/* FUNCTION : CEEMOUT - dispatch a message to message file   */
/*          : CEENCOD - construct a condition token          */
/*          : CEEMGET - retrieve, format and store a message */
/*          : CEEDCOD - decode an existing condition token    */
/*          : CEEMSG  - retrieve, format, and dispatch a      */
/*                      message to message file              */
/*                                                           */
/*   This example illustrates the invocation of the Language */
/*   Environment message and condition handling services.   */
/*   It constructs a condition token, retrieves the associated */
/*   message, and outputs the message to the message file.  */
/*                                                           */
/*   This example program outputs the Language Environment   */
/*   message, "CEE0260S"                                     */
/*                                                           */
/*************************************************************/
DCL MSGSTR CHAR(255) VARYING;
DCL MSGDEST REAL FIXED BINARY(31,0);
DCL MSGNUM REAL FIXED BINARY(15,0);
DCL CASE REAL FIXED BINARY(15,0);
DCL SEV REAL FIXED BINARY(15,0);
DCL SEV2 REAL FIXED BINARY(15,0);
DCL CNTRL REAL FIXED BINARY(15,0);
DCL FACID CHARACTER (3);
DCL ISINFO REAL FIXED BINARY(31,0);
DCL MSGINDX REAL FIXED BINARY(31,0);
DCL 01 CTOK,                   /* Feedback token */
   03 MsgSev    REAL FIXED BINARY(15,0),
   03 MsgNo     REAL FIXED BINARY(15,0),
   03 Flags,    /* Feedback token */
   05 Case      BIT(2),
   05 Severity  BIT(3),
   05 Control   BIT(3),
   03 FacID     CHAR(3),    /* Facility ID */
   03 ISI       /* Instance-Specific Information */
   REAL FIXED BINARY(31,0);
DCL 01 FC,                      /* Feedback token */
   03 MsgSev    REAL FIXED BINARY(15,0),
   03 MsgNo     REAL FIXED BINARY(15,0),
   03 Flags,    /* Feedback token */
   05 Case      BIT(2),
   05 Severity  BIT(3),
   05 Control   BIT(3),
03 FacID CHAR(3), /* Facility ID */
03 ISI /* Instance-Specific Information */
REAL FIXED BINARY(31,0); DCL 01 MGETFC.

/* Feedback token */
03 MsgSev REAL FIXED BINARY(15,0),
03 MsgNo REAL FIXED BINARY(15,0),
03 Flags,
  05 Case BIT(2),
  05 Severity BIT(3),
  05 Control BIT(3),
03 FacID CHAR(3), /* Facility ID */
03 ISI /* Instance-Specific Information */
REAL FIXED BINARY(31,0);
DCL MSGAREA CHAR(80);

PUT SKIP LIST('PL/I message example is now in motion');
MSGSTR = 'The following message, CEE0260S, is expected';
MSGDEST = 2;
/*************************************************************/
/*   Call CEEMOUT to output informational message.       */
/*   Call CEEMSG to output error message if CEEMOUT fails.*/
/*************************************************************/
CALL CEEMOUT ( MSGSTR, MSGDEST, FC );
IF ¬ FBCHECK( FC, CEE000 )  THEN
  CALL CEEMSG( FC, MSGDEST, MGETFC );
/*************************************************************/
/*   Set up token fields for creation of a condition token*/
/*************************************************************/
SEV = 3;
MSGNUM = 260;
CASE = 1;
CNTRL = 1;
FACID = 'CEE';
ISINFO = 0;
/*************************************************************/
/*   Call CEENCED to construct a condition token        */
/*************************************************************/
CALL CEENCED ( SEV, MSGNUM, CASE, SEV2, CNTRL, FACID, ISINFO, CTOK, FC );
IF ¬ FBCHECK( FC, CEE000)   THEN  DO;
  MSGINDEX = 0;
  MSGAREA = ' '; /* Call CEEMGET to retrieve message 260. Since */
  /* message 260 is longer than the length of MSGAREA, */
  /* a DO UNTIL statement loop is used to call CEEMGET */
  /* multiple times until the message index is zero. */
  /*************************************************************/
Retrieve_Message:
DO UNTIL( MSGINDEX = 0 );
  CALL CEEMGET( CTOK, MSGAREA, MSGINDEX, MGETFC );
  IF ¬ FBCHC0CK( MGETFC, CEE000) THEN DO;
    /*************************************************************/
    /*Call CEEDCOD to decode CEEMGET's feedback token */
    /*************************************************************/
    CALL CEEDCOD ( MGETFC, SEV, MSGNUM, CASE, SEV2, CNTRL, FACID, ISINFO, FC );
    IF ¬ FBCHC0CK( FC, CEE000) THEN DO;
      /*************************************************************/
      /* Call CEEMSG to output the error message */
      /*************************************************************/
      CALL CEEMSG( MGETFC, MSGDEST, FC );
      PUT SKIP LIST('Error ' || FC.MsgNo
                   || ' from CEEMSG');
      STOP;
    END;
  END;
  /* If decoded message number is not 455, */
  /* then CEEMGET actually failed with error. */
  IF ( MGETFC.MsgNo ¬= 455) THEN DO;
    PUT SKIP LIST( 'Error ' || MGETFC.MsgNo
                   || ' retrieving message CEE0260S');
    STOP;
  END;
END;
/*************************************************************/
/* Call CEEMOUT to output each portion of message 260 */
MSGSTR = MSGAREA;
CALL CEEMOUT (MSGSTR, MSGDEST, FC);
IF (MSGINDX = 0) THEN DO;
   PUT SKIP LIST ('******************************************************');
   PUT SKIP LIST ('PL/I message example program ended');
   PUT SKIP LIST ('******************************************************');
   END;
END Retrieve_Message /* END DO UNTIL MSGINDX = 0 */;
END /* CEENCOD successful */;
ELSE DO;
   PUT SKIP LIST ('Error ' || FC.MsgNo || ' in encoding condition token');
   END;
END CE92MSG;
Chapter 20. Using date and time services

This topic describes Language Environment date and time services and includes examples showing calls to those services.

The basics of using date and time services

Language Environment includes a complete set of callable services that help HLLs perform date and time calculations. You can use these services to read, calculate, and write values representing the date and time. Language Environment offers unique pattern-matching capabilities that let you process almost any date and time format contained in an input record or produced by operating system services.

You can use date and time services to:

- Format date and time values by country code
- Format date and time values using customized formats
- Parse date values and time values
- Convert between Gregorian, Julian, Asian, and Lilian formats
- Calculate days between dates
- Calculate elapsed time to the nearest millisecond
- Get local time and Greenwich Mean Time (GMT) from the system without a supervisor call (SVC) overhead
- Properly handle 2-digit years in the year 2000

All Language Environment date and time services are enabled for national language support, including full DBCS support for the Japanese Emperor era. For more information about national language support, see Chapter 21, “National language support,” on page 311.

All Language Environment date and time services are based on the Gregorian calendar, with Lilian limits as described in “Date limits” on page 286.

Related services

Callable services

**CEECBLDY**
Converts character date value to the COBOL Integer format. Day one is 01 January 1601 and the value is incremented by one for each subsequent day. This service is similar to CEEDAYS, except that it provides an answer in COBOL Integer format, so that it is compatible with ANSI COBOL intrinsic functions. It should not be used with other Language Environment date or time services.

**CEEDATE**
Converts dates in the Lilian format to character values

**CEEDATM**
Converts number of seconds to character timestamp

**CEEDAYS**
Converts character date values to the Lilian format. Day one is 15 October 1582, and the value is incremented by one for each subsequent day.

**CEEDYWK**
Provides day of week calculation

**CEEGMT**
Gets current Greenwich Mean Time (date and time)
**CEEGMTO**
Gets difference between Greenwich Mean Time and local time

**CEEISEC**
Converts binary year, month, day, hour, minute, second, and millisecond to a number representing the number of seconds since 00:00:00 14 October 1582

**CEELOCT**
Gets current date and time

**CEEQCEN**
Queries the century window

**CEESCEN**
Sets the century window

**CEESECI**
Converts a number representing the number of seconds since 00:00:00 14 October 1582 to seven separate binary integers representing year, month, day, hour, minute, second, and millisecond

**CEESECS**
Converts character timestamps (a date and time) to the number of seconds since 00:00:00 14 October 1582

**CEEUTC**
Same as CEEGMT

See *z/OS Language Environment Programming Reference* for syntax and examples of these callable services.

### Working with date and time services

Before you can start working with date and time services, you need to know the various formats for specifying date and times and any limits that exist.

#### Date limits

All Language Environment date and time services are based on the Gregorian calendar, which has certain limits for the date variables. These limits are:

**Starting Lilian Date**
The beginning of the valid Lilian date range (day one) is Friday, 15 October 1582, the date the Gregorian calendar was adopted. Lilian dates preceding this date are undefined. In the Lilian date range:

- Day zero equals 00:00:00 14 October 1582.
- Day one equals 00:00:00 15 October 1582.

All valid Lilian dates must be after 00:00:00 15 October 1582.

**Starting COBOL Integer Date (ANSI COBOL Intrinsic Functions)**
The beginning of the COBOL Integer date range according to the COBOL standard is 31 December 1600. COBOL Integer dates preceding this date are undefined. In the COBOL Integer date range:

- Day zero equals 00:00:00 31 December 1600.
- Day one equals 00:00:00 01 January 1601.

All valid COBOL Integer dates must be after 00:00:00 01 January 1601.

COBOL has a compiler option, INTDATE, that allows you to get and use Lilian integer dates with COBOL Intrinsic Functions or to use the ANSI starting dates. Use INTDATE(LILIAN) if you want to pass integer dates between programs of different languages and use both Intrinsic Functions and Language Environment callable services to process the integer dates.
End Lilian Date (End COBOL Integer Date)

The end of the Lilian date range, as well as the COBOL Integer date range, is set to 31 December 9999. Lilian dates and COBOL Integer dates following this date are undefined.

Limit of Current Era

The maximum future date you can express in an era system must be within the first 999 years of the current era. Future dates past year 999 of the current era are undefined.

Picture character terms and picture strings

Picture character terms define the format of date and time fields. A picture string is a template that indicates the format of the input data. For example, the format of the date 06/16/1990 (where 06 is the month, 16 is the day, and 1990 is the year) corresponds to the picture string MM/DD/YYYY. See z/OS Language Environment Programming Reference for the Language Environment picture character term and picture string values.

Notation for eras

Calendars based on eras use unique picture strings to identify the eras. The era picture string begins with a less than character (<) and ends with the greater than character (>). The characters between the less than and greater than characters are the era name in DBCS characters.

Japanese Era

The six-character string <JJJJ>. An example of specifying the Japanese Meiji era would be to specify X'0E45A645840F' where the X'0E' and X'0F' are the less than character (<) and greater than character (>), respectively. Refer to z/OS Language Environment Programming Reference for the Japanese eras used by Language Environment date and time services.

Performing calculations on date and time values

Language Environment stores a date as a fullword binary integer and a timestamp as a doubleword floating-point value. You can use these formats to perform arithmetic calculations on date and time values, instead of writing special subroutines to do so. Figure 88 on page 287 is an example of how you can use Language Environment date and time services to convert a date to a different format and perform a simple calculation on the formatted date.

In this example, the number of years of service for an employee is determined using the original date of hire in the format YYMMDD to make the calculations. The example calculates the total number of years of service for an employee by first calling CEEDAYS to convert the days to Lilian and by then calling CEELOCT (Get Current Local Time) to get the current local time. Then, doh_Lilian is subtracted from today_Lilian (the number of days from the beginning of the Gregorian calendar to the current local time) to calculate the employee’s total number of days of employment. The final calculation divides that number by 365.25 to get the number of service years.

```
CALL CEEDAYS (date_of_hire, 'YYMMDD', doh_lilian, fc)
CALL CEELOCT (today_Lilian, today_seconds, today_Gregorian, fc)
service_days = today_Lilian - doh_Lilian
service_years = service_days / 365.25
```

Figure 88: Performing calculations on dates

The valid Lilian date range is 15 October 1582 to 31 December 9999. However, COBOL intrinsic functions uses the COBOL Integer date 01 January 1601 as day one. Language Environment provides the CEECBLDY callable service to allow you to work with the COBOL Integer date format. See z/OS Language Environment Programming Reference for more information about the CEECBLDY service.
Century window routines

To process 2-digit years in the year 2000 and beyond, Language Environment employs a sliding scheme called a century window where all 2-digit years lie within a 100-year interval. The default century window for Language Environment is set to start 80 years before the current system date. In the following example, 1993 is the current system date. The century window spans one hundred years from 1913 to 2012 where years 13 through 99 are recognized as 1913-1999 and years 00 through 12 are recognized as 2000-2012.

<table>
<thead>
<tr>
<th>1913</th>
<th>1993</th>
<th>2012</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

System Date

Figure 89: Default century window

In 1994, years 14 through 99 are recognized as 1914-1999, and years 00 through 13 are recognized as 2000-2013. By year 2080, all 2-digit years would be recognized as 20xx. In 2081, 00 would be recognized as year 2100.

Some applications might need to set up a different 100-year interval. For example, banks often deal with 30-year bonds, which could be due 01/31/20. You can use the CEESCEN callable service (see z/OS Language Environment Programming Reference) to change the century window. For example, the following statement sets the default century to the 100-year interval starting 30 years before the system date, instead of the Language Environment default of 80 years:

Call CEESCEN(30, fc)

Call CEESCEN(30, fc)

<table>
<thead>
<tr>
<th>1963</th>
<th>1993</th>
<th>2062</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

System Date

Figure 90: Using CEESCEN to change the century window

A companion service, CEEQCEN, queries the current century window. A subroutine can, for example, use a different interval for date processing than the parent routine. Before returning, the subroutine resets the interval back to its previous value. For more information about changing the century window, see “Examples illustrating calls to CEEQCEN and CEESCEN” on page 289.

National Language Support for date and time services

The NATLANG and COUNTRY runtime options provide national language support for date and time services. The names of the months and days of the week are based on the national language specified in the NATLANG option. Some date and time services also allow the specification of a blank or null picture string, a practice that directs Language Environment to use a date and time format based upon the current value specified in the COUNTRY option. You can locate the default date and time format for any supported country by using the CEEFMDA, CEEFMDT, or CEEFMTM callable services.
Examples using date and time callable services

The examples in this topic illustrate some of the date conversion and manipulation you can perform by using the Language Environment date and time services together. There are examples for the following services:

**CEEQCEN**
Queries the century window. See “Examples illustrating calls to CEEQCEN and CEESCEN” on page 289.

**CEESCEN**
Sets the century window. See “Examples illustrating calls to CEEQCEN and CEESCEN” on page 289.

**CEESECS**
Converts timestamp to seconds. See “Examples illustrating calls to CEESCECS” on page 291.

**CEESECS and CEEDATM**
Converts timestamp to seconds and builds a new timestamp. See “Examples illustrating calls to CEESECS and CEEDATM” on page 294.

**CEESECS, CEESECI, CEEISEC, and CEEDATM**
Converts timestamp to seconds, convert seconds to date and time components, convert date and time to seconds, and build new timestamp (see “Examples illustrating calls to CEESECS, CEESECI, CEEISEC, and CEEDATM” on page 298).

**CEEDAYS, CEEDYWK, and CEEDATE**
Converts a date to a Lilian date, converts Lilian date to calendar format, and returns day of week for the derived Lilian date. See “Examples illustrating calls to CEEDAYS, CEEDATE, and CEEDYWK” on page 303.

**CEECBLDY**
Converts a date to a COBOL Integer date that is compatible with ANSI COBOL intrinsic functions. See “Calls to CEECBLDY in COBOL” on page 308.

Examples illustrating calls to CEEQCEN and CEESCEN

The following topics contain examples to illustrate how to query the current century window and how to set a new window with a new default of 30 years.

**Calls to CEEQCEN and CEESCEN in C or C++**

```c
/*Module/File Name:  EDCCWIN */
/**************************************************************************/
/* Demonstrates how to use CEEQCEN and CEESCEN to query and            */
/* set the century window.                                             */
/**************************************************************************/
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <leawi.h>
#include <ceeedcct.h>
int main (void) {
  _INT4 oldcen, tempcen;
  _FEEDBACK qcenfc, scenfc;
  /* Call CEEQCEN to retrieve and save current century window */
  CEEQCEN ( &oldcen , &qcenfc );
  if ( _FBCHECK ( qcenfc , CEE000 ) != 0 ) {
    printf("CEEQCEN failed with message number %d\n", qcenfc.tok_msgno);
    exit(1999);
  }
  /* Call CEESCEN to temporarily change century window to 30 */
  tempcen = 30;
  CEESCEN ( &tempcen , &scenfc );
  if ( _FBCHECK ( scenfc , CEE000 ) != 0 ) {
```
printf("CEESCEN (1st call) failed with message number %d\n", scenfc.tok_msgno);
exit(2999);
}

/* Perform date processing with 2-digit years... */
/* Call CEESCEN again to reset century window */
CEESCEN(&oldcen , &scenfc);
if (_FBCHECK(_FBCECN , CEE000) != 0) {
printf("CEESCEN (2nd call) failed with message number %d\n", scenfc.tok_msgno);
exit(3999);
}
exit(0);

Calls to CEEQCEN and CEESCEN in COBOL

CBL LIB,QUOTE
*Module/File Name: IGZTCWIN
******************************************************************************
* Demonstrates how to use CEEQCEN and CEESCEN to query and set the century window. *
******************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. CBCENTW.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 OLDCEN PIC S9(9) BINARY.
77 TEMPcen PIC S9(9) BINARY.
01 FC.
 02 Condition-Token-Value.
COPY CEEIQZCT.
 03 Case-1-Condition-ID.
    04 Severity PIC S9(4) BINARY.
    04 Msg-No PIC S9(4) BINARY.
 03 Case-2-Condition-ID
    REDEFINES Case-1-Condition-ID.
    04 Class-Code PIC S9(4) BINARY.
    04 Cause-Code PIC S9(4) BINARY.
 03 Case-Sev-Ctl PIC X.
 03 Facility-ID PIC XXX.
 02 I-S-Info PIC S9(9) BINARY.
PROCEDURE DIVISION.
******************************************************************************
** Call CEEQCEN to retrieve and save current century window **
******************************************************************************
CALL "CEEQCEN" USING OLDCEN , FC.
IF NOT CEE000 of FC THEN
DISPLAY "CEEQCEN failed with msg "
Msg-No of FC UPON CONSOLE
STOP RUN
END-IF.
******************************************************************************
** Call CEESCEN to temporarily change century window to 30 **
******************************************************************************
MOVE 30 TO TEMPcen.
CALL "CEESCEN" USING TEMPcen , FC.
IF NOT CEE000 of FC THEN
DISPLAY "First call to CEESCEN failed with msg "
Msg-No of FC UPON CONSOLE
STOP RUN
END-IF.

** Perform date processing with 2-digit years... **
  ** Call CEESCEN again to reset century window **
CALL "CEESCEN" USING OLDCEN , FC.
IF NOT CEE000 of FC THEN
DISPLAY "Second call to CEESCEN failed with msg "
Msg-No of FC UPON CONSOLE
STOP RUN
END-IF.
GOBACK.
Examples illustrating calls to CEESECS

The following examples illustrate calls to CEESECS to compute the total number of hours between two timestamps.

Calls to CEESECS in C or C++

```c
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>

main ()
{
    _VSTRING StartTime;
    /*...*/
} /*...*/
```
**Calls to CEESecs in COBOL**

CBL LIB,QUOTE

*Module/File Name: IGZTDT1*

******************************************************************************
**
** CEE78DAT - Call CEESecs to convert timestamp to seconds
**
** This example calls the LE CEESecs callable
**
** service to compute the number of hours between
**
** the timestamps 11/02/92 05:22 and 11/02/92 17:22. **
**
** The program responds that 36 hours has elapsed. **
**
******************************************************************************

IDENTIFICATION DIVISION.
PROGRAM-ID. CE78DAT.
DATA DIVISION.
WORKING-STORAGE SECTION.
* Double precision is needed for the seconds results

```
01 START-SECS COMP-2.
01 END-SECS COMP-2.
01 EOF-SWITCH PIC X VALUE "N".
  88 EOF VALUE "Y".
01 FC.
  02 Condition-Token-Value.
COPY CEEIGZCT.
03 Case-1-Condition-ID.
  04 Severity PIC S9(4) BINARY.
  04 Msg-No PIC S9(4) BINARY.
03 Case-2-Condition-ID
```
REDEFINES Case-1-Condition-ID.
04 Class-Code      PIC S9(4) BINARY.
04 Cause-Code      PIC S9(4) BINARY.
03 Case-Sev-Ctl    PIC X.
03 Facility-ID     PIC XXX.
02 I-S-Info        PIC S9(9) BINARY.
01 PICSTR.
  02 Vstring-length PIC S9(4) BINARY.
  02 Vstring-text.
    03 Vstring-char PIC X
      OCCURS 0 TO 256 TIMES
      DEPENDING ON Vstring-length
      of PICSTR.
  01 START-TIME.
    02 Vstring-length PIC S9(4) BINARY.
    02 Vstring-text.
      03 Vstring-char PIC X
        OCCURS 0 TO 256 TIMES
        DEPENDING ON Vstring-length
        of START-TIME.
  01 END-TIME.
    02 Vstring-length PIC S9(4) BINARY.
    02 Vstring-text.
      03 Vstring-char PIC X
        OCCURS 0 TO 256 TIMES
        DEPENDING ON Vstring-length
        of END-TIME.
  01 INPUT-VARIABLES.
    05 ELAPSED-TIME        PIC S9(5)V99 PACKED-DECIMAL.
    05 ELAPSED-TIME-OUT    PIC +Z(4)9.99.

PROCEDURE DIVISION.
0001-BEGIN-PROCESSING.
  MOVE 14 TO Vstring-length of PICSTR.
  MOVE "MM/DD/YY HH:MI" TO Vstring-text of PICSTR.
  MOVE 14 TO Vstring-length of START-TIME.
  MOVE "11/02/92 05:22" TO Vstring-text of START-TIME.
  MOVE 14 TO Vstring-length of END-TIME.
  MOVE "11/03/92 17:22" TO Vstring-text of END-TIME.
  * *********************************************************
  * CEESECS takes the timestamp START-TIME and returns a *
  * double-precision Lilian seconds tally in START-SECS. *
  * *********************************************************
  CALL "CEESECS" USING START-TIME, PICSTR, START-SECS, FC
  IF CEE000 of FC THEN
  * *********************************************************
  * CEESECS takes the timestamp END-TIME and returns a *
  * double-precision Lilian seconds tally in END-SECS. *
  * *********************************************************
  CALL "CEESECS" USING END-TIME, PICSTR, END-SECS, FC
  IF CEE000 of FC THEN
    COMPUTE ELAPSED-TIME = (END-SECS - START-SECS) / 3600
    MOVE ELAPSED-TIME TO ELAPSED-TIME-OUT
    DISPLAY ELAPSED-TIME-OUT
      " hours have elapsed between "
      Vstring-text of START-TIME
      " and " Vstring-text of END-TIME
  ELSE
    DISPLAY "Error " Msg-No of FC
    " converting ending date to Lilian date"
    STOP RUN
  END-IF
  ELSE
    DISPLAY "Error " Msg-No of FC
    " converting starting date to Lilian date"
    STOP RUN
  END-IF
  GOBACK.

Calls to CEESECS in PL/I

*PROCESS MACRO;
/*Module/File Name: IBMDT1 */
/****************************************************************************/
/* Function: CEESECS - convert timestamp to seconds */
/****************************************************************************/
/* This example calls the CEESECS callable */
Examples illustrating calls to CEESECS and CEEDATM

The following examples illustrate calls to date and time services to convert a timestamp to seconds (CEESECS), twenty-four hours in seconds is subtracted from the original timestamp value, and a new timestamp is built (CEEDATM) for the updated number of seconds.

Calls to CEESECS and CEEDATM in C or C++

```c
/*Module/File Name:  EDCDT2   */
/**********************************************************/
/*                                                        */
/*Function : CEESECS - convert timestamp to seconds */
/* : CEEDATM - convert seconds to timestamp */
/* :                                        */
/*CEESECS is used to convert a timestamp to seconds. */
/*24 hours in seconds is subtracted from */
```
/* the number of seconds in the original timestamp. */
/*CEEDATM is then used to build a new timestamp */
/*representing the new date and time, 11/01/92 05:22. */
/* */
/***********************************************************/
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>
#define TimeStamp "11/02/92 05:22"
#define displacement 24
main ()
{
  int User_Input();
  _VSTRING Time_Stamp;
  _CHAR80 New_TimeStamp;
  _VSTRING  picstr;
  _FLOAT8 Lilian_Seconds;
  _FLOAT8 New_Secs;
  _FEEDBACK FC;
  _INT4 dest=2;
  char  New_Time[15];

  /*********************************************************************/
The date picstr must be set to match the timestamp format.
strncpy (picstr.string,"MM/DD/YY HH:MI",14);
picstr.length = 14;
/*********************************************************************/
/* In the following loop the timestamp is converted to Lilian */
/* seconds. 24 hours in seconds are subtracted from the */
/* Lilian seconds and a new timestamp is created and */
/* displayed. */
/*********************************************************************/
strncpy(Time_Stamp.string,TimeStamp,14);
Time_Stamp.length = 14;
/*********************************************************************/
CEESECS takes the user-entered timestamp Time_Stamp and
returns a double-precision Lilian seconds tally in
Lilian_Seconds
*********************************************************************/
CEESECS ( &Time_Stamp, &picstr , &Lilian_Seconds , &FC );
if ( (_FBCHECK (FC , CEE000)) == 0 )
{
  /*********************************************************************/
The displacement variable is subtracted from the Lilian
seconds tally in Lilian_Seconds
*********************************************************************/
New_Secs = Lilian_Seconds - displacement * 3600.0;
ASSWORD*********************************************************************/
CEEDATM is invoked to get a new timestamp value based on the
new Lilian seconds tally in New_Secs.
*********************************************************************/
CEEDATM ( &New_Secs, &picstr , New_TimeStamp , &FC );
if ( (_FBCHECK (FC , CEE000)) == 0 )
{
  New_TimeStamp[14] = '\0';
sprintf(New_Time,'%s\0',New_TimeStamp);
  printf("%s is the time %i hours before %s
",
              New_Time, displacement, TimeStamp);
}
else
{
  printf ( "Error converting Seconds to TimeStamp.\n" );
  CEEMSG(&FC, &dest, NULL);
}
else
{
  printf ( "Error converting TimeStamp to seconds.\n" );
  CEEMSG(&FC, &dest, NULL);
}
}
** CEE80DAT - Call CEESECS to convert timestamp to seconds**
** and CEEDATM to convert seconds to timestamp **
** CEESECS is used to convert a timestamp to seconds. **
** 24 hours in seconds is subtracted from **
** the number of seconds in the original timestamp. **
** CEEDATM is then used to build a new timestamp for **
** the updated number of seconds. **

********************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. CE80DAT.
DATA DIVISION.
WORKING-STORAGE SECTION.
* Double precision needed for the seconds results
01 START-SECS COMP-2.
01 NEW-TIME COMP-2.
01 FC.*
  02 Condition-Token-Value.
COPY CEEIGZCT.
  03 Case-1-Condition-ID.
   04 Severity PIC S9(4) BINARY.
   04 Msg-No PIC S9(4) BINARY.
   03 Case-2-Condition-ID
   REDEFINES Case-1-Condition-ID.
   04 Class-Code PIC S9(4) BINARY.
   04 Cause-Code PIC S9(4) BINARY.
   03 Case-Seq-Ctl PIC X.
  03 Facility-ID PIC XXX.
  02 I-S-Info PIC S9(9) BINARY.
01 PICSTR.
  02 Vstring-length PIC S9(4) BINARY.
  02 Vstring-text.
    03 Vstring-char PIC X
    OCCURS 0 TO 256 TIMES
    DEPENDING ON Vstring-length of PICSTR.
 01 WS-TIMESTAMP.
  02 Vstring-length PIC S9(4) BINARY.
  02 Vstring-text.
    03 Vstring-char PIC X
    OCCURS 0 TO 256 TIMES
    DEPENDING ON Vstring-length of WS-TIMESTAMP.
01 NEW-TIMESTAMP PIC X(80).
01 INPUT-VARIABLES.
  05 SECONDS-DISPLACED PIC S9(9) BINARY.
  05 ELAPSED-TIME-OUT PIC +Z(4)9.99.

PROCEDURE DIVISION.
0001-BEGIN-PROCESSING.
MOVE 14 TO Vstring-length of PICSTR.
MOVE "MM/DD/YY HH:MI" TO Vstring-text of PICSTR.
MOVE 14 TO Vstring-length of WS-TIMESTAMP.
* **********************************************************
* CEESECS is invoked to obtain the Lilian seconds tally *
* corresponding to the timestamp 11/02/92 05:22.         *
* **********************************************************
CALL "CEESECS" USING WS-TIMESTAMP, PICSTR, START-SECS, FC.
IF CEE000 of FC THEN
* **********************************************************
* The Lilian seconds tally in START-SECS is *
* decremented by 24 hours worth of seconds.. *
* **********************************************************
CREATE NEW-TIME = START-SECS - 24 * 3600.
* **********************************************************
* CEEDATM is invoked to obtain a new timestamp *
* based on the new Lilian seconds tally. *
* **********************************************************
CALL "CEEDATM" USING NEW-TIME, PICSTR, NEW-TIMESTAMP, FC
IF CEE000 of FC THEN
  DISPLAY "The time 24 hours before "
  Vstring-text of WS-TIMESTAMP
  " is " NEW-TIMESTAMP
ELSE
  DISPLAY "Error converting seconds to timestamp."
STOP RUN
END-IF
Calls to CEESECS and CEEDATM in PL/I

*PROCESS MACRO;
/*Module/File Name: IBMDT2 */
/****************************************************/
/*                                                  */
/* Function: CEESECS - convert timestamp to seconds */
/*         : CEEDATM - convert seconds to timestamp */
/*                                                  */
/* CEESECS is used to convert a timestamp to */
/* seconds. 24 hours in seconds is subtracted from */
/* the number of seconds in the original timestamp. */
/* CEEDATM is then used to build a new timestamp */
/* representing the new date and time. */
/*                                                  */
/****************************************************/
PLIDS: PROC OPTIONS(MAIN);
%INCLUDE CEEIBMAW;
%INCLUDE CEEIBMCT;
DCL TIMESTAMP CHAR(255) VARYING INIT('01/26/67 20:00');
DCL NEW_TIMESTAMP CHAR(80);
DCL PICSTR CHAR(255) VARYING INIT('MM/DD/YY HH:MI');
DCL START_SECS REAL FLOAT DECIMAL(16);
DCL NEW_TIME REAL FLOAT DECIMAL(16);
DCL DISPLACEMENT REAL FIXED BINARY(31,0) INIT(24);
DCL 01 FC,                     /* Feedback token */
03 MsgSev REAL FIXED BINARY(15,0),
03 MsgNo REAL FIXED BINARY(15,0),
03 Flags,          */
05 Case BIT(2), 05 Severity BIT(3),
05 Control BIT(3),
03 FacID /* Facility ID */
03 ISI /* Instance-Specific Information */
    REAL FIXED BINARY(31,0);
/****************************************************/
/* CEESECS is invoked to obtain the Lilian */
/* seconds tally corresponding to the timestamp */
/* 01/26/67 20:00. The Lilian seconds tally is */
/* returned in double-precision variable */
/* START_SECS. */
/****************************************************/
CALL CEESECS ( TIMESTAMP, PICSTR, START_SECS, FC );
IF FBCHECK( FC, CEE000) THEN DO;
/****************************************************/
/* The Lilian seconds tally in START_SECS is */
/* decremented by 24 hour DISPLACEMENT */
/* variable times 3600 seconds. */
/****************************************************/
NEW_TIME = START_SECS - DISPLACEMENT * 3600;
/****************************************************/
/* CEEDATM is invoked to obtain a new */
/* TimeStamp based on the new Lilian seconds. */
/****************************************************/
CALL CEEDATM ( NEW_TIME, PICSTR, NEW_TIMESTAMP, FC );
IF FBCHECK( FC, CEE000) THEN DO;
   PUT SKIP LIST( 'The time ' || DISPLACEMENT */
   ' hours before ' || TIMESTAMP */
   ' is ' || NEW_TIMESTAMP );
END;
ELSE DO;
   PUT SKIP LIST('ERROR CONVERTING SECONDS TO TIMESTAMP');
   PUT SKIP LIST( 'CEEDATM failed with msg '|| FC.MsgNo );
END;
END;
ELSE DO;
   PUT SKIP LIST('ERROR CONVERTING TIMESTAMP TO SECONDS');
   PUT SKIP LIST( 'CEESECS failed with msg '|| FC.MsgNo );
END;
Examples illustrating calls to CEESECS, CEESECI, CEEISEC, and CEEDATM

The following examples illustrate calls to date and time services to convert a timestamp into seconds (CEESECS), convert the seconds to a date and time component (CEESECI), add thirty-two months to the month component, convert the date and time component back to seconds (CEEISEC), and build a new timestamp (CEEDATM).

Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in C or C++

```c
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>
#define TimeStamp "11/02/92 05:22"
#define displacement 32
void main ()
{
    _VSTRING Time_Stamp;
    _CHAR80 New_TimeStamp;
    _VSTRING picstr;
    _FLOAT8 Lilian_Seconds;
    _FLOAT8 New_Secs;
    _FEEDBACK FC;
    char New_Time[15];
    int Month_in_Century;
    //Date/time components for CEESECI, CEEISEC.
    _INT4 year;
    _INT4 month;
    _INT4 days;
    _INT4 hours;
    _INT4 minutes;
    _INT4 seconds;
    _INT4 millsec;
    /*The date picstr must be set to match the timestamp format.
    strcpy (picstr.string,"MM/DD/YY HH:MI");
    picstr.length = 14;
    strncpy(Time_Stamp.string,TimeStamp,14);
    Time_Stamp.length = 14;
    */
    /*CEESECS takes the timestamp "11/02/92 05:22" and returns a double-precision Lilian seconds tally in Lilian_Seconds
    CEESECS ( &Time_Stamp, &picstr , &Lilian_Seconds , &FC );
    if ((_FBCHECK (FC, CEE000)) == 0)
    {
        //CEESECI converts the Lilian seconds tally in Lilian_Seconds and returns date/time components.
        CEESECI ( &Lilian_Seconds, &year, &month, &days, &hours, &minutes, &seconds, &millsec, &FC );
        if ((_FBCHECK (FC, CEE000)) == 0)
        {
            //CEEDATM is then used to build a new timestamp for the new time.
            CEEDATM ( &New_TimeStamp, &picstr, &year, &month, &days, &hours, &minutes, &seconds, &millsec, &FC );
            if ((_FBCHECK (FC, CEE000)) == 0)
            {
                //print the new timestamp
                printf("%s
", New_TimeStamp.string);
            }
            else
            {
                printf("Error in CEEDATM
");
            }
        }
        else
        {
            printf("Error in CEESECI
");
        }
    }
    else
    {
        printf("Error in CEESECS
");
    }
}
```

Date and time services

298 z/OS: Language Environment Programming Guide
The month component of the timestamp is converted to month-in-century. Then a new month and a new year are computed from the new month-in-century number. The month date/time component has a range between 1 and 12.

```c
Month_in_Century = year*12 + month + displacement - 1;
year = Month_in_Century / 12;
month = (Month_in_Century % 12) + 1;
```

The month date/time component has been shifted forward 32 months. Our examples gets a new Lilian seconds tally based on the new month and year components. This is done with a call to function CEEISEC. The new Lilian seconds tally is placed in the double-precision variable Lilian_Seconds.

```c
CEEISEC (&year, &month, &days, &hours, &minutes, &seconds, &millsec, &Lilian_Seconds, &FC);
```

If ((_FBCHECK (FC, CEE000)) == 0)

```c
CEEDATM ( &Lilian_Seconds, &picstr, New_TimeStamp, &FC);
```

If ((_FBCHECK (FC, CEE000)) == 0)

```c
New_TimeStamp[14] = '\0';
sprintf(New_Time, "%s\0", New_TimeStamp);
if ( displacement < 0 )
    printf("%s is the time %d months before %s.\n", New_Time, displacement, TimeStamp);
else
    printf("%s will be the time %d months after %s.\n", New_Time, displacement, TimeStamp);
}
else
    printf ( "Error converting Seconds to TimeStamp.\n" );
else
    printf ( "Error converting Components to seconds.\n" );
else
    printf ( "Error converting seconds to components.\n" );
else
    printf ( "Error converting TimeStamp to seconds\n" );
}
```

Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in COBOL

CBL LIB,QUOTE

**Module/File Name: IGZTDT3**

******************************************************************************
* CE81DATA - Call the following LE service routines: *
* CEESECS - convert timestamp to seconds *
* CEESECI - convert seconds to time components *
* CEEISEC - convert time components to seconds *
* CEEDATM - convert seconds to timestamp *
* CEESECS is used to convert the timestamp to seconds *
* CEESECI is used to convert seconds to date/time components.*
* 32 months is added to the month and year component *
* of date/time. *
* CEEISEC is to convert the date/time components with the *
* new months component back to a Lilian seconds tally. *
* CEEDATM is then used to build a new timestamp for *
* the updated number of seconds. *
******************************************************************************

IDENTIFICATION DIVISION.
PROGRAM-ID. CEE81DAT.
DATA DIVISION.
WORKING-STORAGE SECTION.
* Double precision needed for the seconds results
01 START-SECS                COMP-2.
01 NEW-TIME                  COMP-2.
01 FC.
  02 Condition-Token-Value.
COPY CEEIGZCT.
    03 Case-1-Condition-ID.
      04 Severity        PIC S9(4) BINARY.
      04 Msg-No          PIC S9(4) BINARY.
    03 Case-2-Condition-ID.
      REDEFINES Case-1-Condition-ID.
      04 Class-Code      PIC S9(4) BINARY.
      04 Cause-Code      PIC S9(4) BINARY.
    03 Case-Sev-Ctl          PIC X.
    03 Facility-ID     PIC XXX.
  02 I-S-Info                  PIC S9(9) BINARY.
01 PICSTR.
  02 Vstring-length   PIC S9(4) BINARY.
  02 Vstring-text.
    03 Vstring-char    PIC X
      OCCURS 0 TO 256 TIMES
      DEPENDING ON Vstring-length
      of PICSTR.
01 WS-TIMESTAMP.
  02 Vstring-length   PIC S9(4) BINARY.
  02 Vstring-text.
    03 Vstring-char    PIC X
      OCCURS 0 TO 256 TIMES
      DEPENDING ON Vstring-length
      of WS-TIMESTAMP.
01 NEW-TIMESTAMP
    PIC X(80).
* ***********************************************************
* These are the date/time variables used by *
* CEEISEC and CEESECI.
* ***********************************************************
* *******************************************************
01 DATE-TIME-COMPONENTS       PIC X.
  05 YEAR                PIC 9(9).
  05 MONTH               PIC 9(9).
  05 DAYS                PIC 9(9).
  05 HOURS               PIC 9(9).
  05 MINUTES             PIC 9(9).
  05 SECONDS             PIC 9(9).
  05 MILLSEC             PIC 9(9).
01 FILLER                  PIC X(80).
01 INPUT-VARIABLES.
  05 MONTHS-TO-DISPLACE  PIC S9(4) BINARY VALUE 32.
  05 DISPLACEMENT-COMP    PIC S9(4) BINARY.
  05 MONTHNUM            PIC 9(9).
PROCEDURE DIVISION.
0001-BEGIN-PROCESSING.
  MOVE 14 TO Vstring-length of WS-TIMESTAMP.
  MOVE "11/02/92 05:22" TO Vstring-text of WS-TIMESTAMP.
  MOVE 14 TO Vstring-length of PICSTR.
  MOVE "MM/DD/YY HH:MI" TO Vstring-text of PICSTR.
* ***********************************************************
* The timestamp "11/02/92 05:22" is converted to *
* seconds under the control of the mask PICSTR. CEESECS *
* will return a Lilian seconds tally in the double- *
* precision floating-point variable START-SECS. *
* ***********************************************************
CALL "CEESECS" USING WS-TIMESTAMP, PICSTR, START-SECS, FC.
  IF CEE000 of FC THEN
  ** ***********************************************************
  ** The Lilian seconds tally in field START-SECS is mapped *
  ** into its date/time components using function CEESECI. *
  ** ***********************************************************
  CALL "CEESECI" USING START-SECS, YEAR, MONTH, DAYS, *
  ** HOURS, MINUTES, SECONDS, MILLSEC, FC *
  ** ***********************************************************
  IF CEE000 of FC THEN
  MOVE MONTHS-TO-DISPLACE TO DISPLACEMENT-COMP
  ** ***********************************************************
  ** MONTH is converted to month-in-century for the *
  ** displacement arithmetic. To a new month and *
  ** year are computed from the new month-in-century *
  ** number (in variable MONTHNUM). The months com- *
  ** ponent has an allowed range of between 1 and 12.*
  ** ***********************************************************
COMPUTE MONTHNUM = YEAR * 12 + MONTH + DISPLACEMENT-COMP - 1
DIVIDE MONTHNUM BY 12 GIVING YEAR REMAINDER MONTH
ADD 1 TO MONTH
* **************************************************
* Now that the MONTH DateTime component has    *
* been shifted forward by 32 months,            *
* we must get a new Lilian seconds tally based *
* on the new MONTH and YEAR components. We    *
* do this with a call to the CEEISEC callable *
* service. The new Lilian seconds tally is    *
* placed in the double-precision field NEW-TIME. *
* **************************************************
CALL "CEEISEC" USING YEAR, MONTH, DAYS, HOURS,
MINUTES, SECONDS, MILLSEC, NEW-TIME, FC
* ******************************************
* CEEDATM is now used to obtain a new    *
* timestamp based on the Lilian seconds *
* tally in the variable New-time.     *
* **************************************************
IF CEE000 THEN
  CALL "CEEDATM" USING NEW-TIME, PICSTR,
  NEW-TIMESTAMP, FC
  IF CEE000 THEN
    DISPLAY "The time "
    MONTHS-TO-DISPLACE " months after "
    Vstring-text of WS-TIMESTAMP
    " is " NEW-TIMESTAMP
  ELSE
    DISPLAY "Error " Msg-No of FC
    " converting seconds to timestamp."
  END-IF
ELSE
  DISPLAY "Error " Msg-No of FC
  " converting components to seconds."
END-IF
ELSE
  DISPLAY "Error " Msg-No of FC
  " converting timestamp to seconds."
END-IF
ELSE
  DISPLAY "Error " Msg-No of FC
  " converting seconds to components."
END-IF
ELSE
  DISPLAY "Error " Msg-No of FC
  " converting components to seconds."
END-IF
ELSE
  DISPLAY "Error " Msg-No of FC
  " converting timestamp to seconds."
END-IF

GOBACK.

Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in PL/I

*PROCESS MACRO;
/*Module/File Name: ISMDT3
 asm
*******************************************************************************/
/* Function      : CEESECS - convert timestamp to seconds */
/* : CEESECI - convert seconds to time components */
/* : CEEISEC - convert time components to seconds */
/* : CEEDATM - convert seconds to timestamp */
/* 32 months is added to the timestamp 11/02/92 05:22 */
/* giving the new timestamp 07/02/95 05:22. */
/* CEESECS is used to convert the timestamp to seconds */
/* CEESECI is used to convert seconds to date/time components */
/* 32 months is added to the month component. */
/* CEEISEC is used to convert the date components to seconds. */
/* CEEDATM is then used to build a new timestamp for the */
/* updated time. */
/********************************************************************************/
CE81DAT: PROC OPTIONS(MAIN);

INCLUDE CEEIBMAW;
INCLUDE CEEIBMCT;

DCL TIMESTAMP       CHAR(255) VARYING INIT('11/02/92 05:22');
DCL NEW_TIMESTAMP   CHAR(80);
DCL PICSTR          CHAR(255) VARYING INIT('MM/DD/YY HH:MI');
DCL START_SECS      REAL FLOAT DECIMAL(16);
DCL NEW_TIME        REAL FLOAT DECIMAL(16);
DCL DISPLACEMENT    FIXED BIN(31,0) INIT(32);
DCL MOUNTHNUM FIXED BIN(31,0);
DCL 01 FC, /* Feedback token */
  03 MsgSev REAL FIXED BINARY(15,0),
  03 MsgNo REAL FIXED BINARY(15,0),
  03 Flags,
  IF 05 Case BIT(2),
    05 Severity BIT(3),
    05 Control BIT(3),
  03 FacID CHAR(3), /* Facility ID */
  03 ISI /* Instance-Specific Information */
    REAL FIXED BINARY(31,0);
/******************************/
/* DATE COMPONENTS FOR CEESECI, CEEISEC */
/******************************/
DCL YEAR REAL FIXED BINARY(31,0);
DCL MONTH REAL FIXED BINARY(31,0);
DCL DAYS REAL FIXED BINARY(31,0);
DCL HOURS REAL FIXED BINARY(31,0);
DCL MINUTES REAL FIXED BINARY(31,0);
DCL SECONDS REAL FIXED BINARY(31,0);
DCL MILLSEC REAL FIXED BINARY(31,0);
/**************************************************************/
/* The timestamp '11/02/92 05:22' is converted to seconds */
/* under the control of the mask PICSTR. CEESECS will */
/* return a Lilian seconds tally in the double-precision */
/* floating-point field START_SECS. */
/**************************************************************/
CALL CEESECS( TIMESTAMP, PICSTR, START_SECS, FC );
IF FBCHECK( FC, CEE000) THEN DO;
/***************************************************************/
/* The Lilian seconds tally in the field START_SECS is mapped */
/* into its date/time components using function CEESECI. */
/***************************************************************/
CALL CEESECI( START_SECS, YEAR, MONTH, DAYS, HOURS, MINUTES,
  SECONDS, MILLSEC, FC);
IF FBCHECK( FC, CEE000) THEN DO;
/***************************************************************/
/* MONTHNUM is converted to month-in-century for the displace- */
/* ment arithmetic. Then a new month and year are computed */
/* from the new month-in-century number. The months */
/* component has an allowed range of between 1 and 12. */
/***************************************************************/
MONTHNUM = YEAR * 12 + MONTH + DISPLACEMENT - 1;
YEAR = MONTHNUM / 12;
MONTH = MOD(MONTHNUM, 12) + 1;
/***************************************************************/
/* Now that the MONTH DateTime component has been shifted */
/* forward by 32 months, we must get a new Lilian seconds tally */
/* based on the new MONTH and YEAR components. */
/* We do this with a call to service CEEISEC. */
/* The new Lilian seconds tally is placed in the double- */
/* precision floating-point variable NEW_TIME. */
/***************************************************************/
CALL CEEISEC( YEAR, MONTH, DAYS, HOURS, MINUTES, SECONDS,
  MILLSEC, NEW_TIME, FC );
IF FBCHECK( FC, CEE000) THEN DO;
/***************************************************************/
/* CEEDATM is now used to obtain a new timestamp based */
/* on the Lilian seconds tally in variable New_time */
/***************************************************************/
CALL CEEDATM( NEW_TIME, PICSTR, NEW_TIMESTAMP, FC );
IF FBCHECK( FC, CEE000) THEN DO;
  PUT SKIP EDIT( 'The time ', DISPLACEMENT,
    ' months after ', TIMESTAMP,
    ' is ', NEW_TIMESTAMP )
    (A, F(4), (3) A);
  END;
ELSE DO;
  PUT SKIP EDIT( 'ERROR ', FC.MsgNo,
    ' CONVERTING SECONDS TO TIMESTAMP')
    (A, F(4), A);
  END;
ELSE DO;
  PUT SKIP EDIT( 'ERROR ', FC.MsgNo,
    ' CONVERTING COMPONENTS TO SECONDS')
    (A, F(4), A);
  END;
ELSE DO;
  PUT SKIP EDIT( 'ERROR ', FC.MsgNo,
Examples illustrating calls to CEEDAYS, CEEDATE, and CEEDYWK

The following examples illustrate calls to date and time services to convert a date to a Lilian date (CEEDAYS). In these examples, a varying number of days are added to the Lilian date, the date is converted back to a character format (CEEDATE), and the day of the week for that Lilian date is returned (CEEDYWK).

Calls to CEEDAYS, CEEDATE, and CEEDYWK for C or C++

```c
/*Module/File Name: EDCDT4 */
/**************************************************************************/
/* Function : CEEDAYS - convert date to Lilian date */
/* : CEEDATE - convert Lilian date to date */
/* : CEEDYWK - find day-of-week from Lilian */
/* CEEDAYS is passed the calander date "11/09/92". The date is originally in YYMMD format and conversion to Lilian format takes place. On return, a varying number of days is added to or subtracted from the Lilian date. CEEDATE is called to convert the Lilian dates to the calendar format "MM/DD/YY".
*/
/* CEEDYWK is called to return the day of the week for each derived Lilian date. */
/* The results are tested for accuracy. */
/* *************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <leawi.h>
#include <ceedcct.h>

char PastFuture;
int NumberOfDays[5] = { 80, 20, 10, 5, 4};
int i;
void main ()
{
    _CHAR80 chrdate;
    _VSTRING  picstr;
    _VSTRING CurrentDate;
    _INT4 Current_Lilian;
    _INT4 Displaced_Lilian;
    _INT4 WeekDay;
    _INT4 ChkWeekDay[5] = { 6, 1, 5, 4, 6 };
    _FEEDBACK FC;
    char Entered_Date[8];
    _INT4 dest=2;
    struct tm *timeptr;
    char Current_Date[6];
    time_t current_time;
    char *ChkDates[] = {
        "08/21/92",
        "11/29/92",
        "11/19/92",
        "11/04/92",
        "11/13/92",
    };

    /*************************************************************/
    /* Set current date to 11/09/92 in YYMMD format */
    /***********************************************************/
    strncpy (CurrentDate.string,"921109",6);
    CurrentDate.length = 6;
```
The date picstr must be adjusted to fit the current date format.

strncpy (picstr.string,"YYMMDD",6);

Call CEEDAYS to convert the date in Current_Date to its corresponding Lilian date format.

CEEDAYS (&CurrentDate, &picstr, &Current_Lilian, &FC);

if ( _FBCHECK (FC , CEE000) != 0 )
{
    printf ("Error in converting current date.\n"");
    CEEMSG(FC, &dest, NULL);
    exit(99);
}

Modify the date picstr to the familiar MM/DD/YY format.

strncpy (picstr.string,"MM/DD/YY",8);

In the following loop, add or subtract the number of days in each element of the NumberOfDays array to the Lilian date. Determine the day of the week for each Lilian date and convert each date back to "MM/DD/YY" format. Issue a message if anything goes wrong.

for (i=0; i < 5; i++)
{
    if (i == 0 || i == 3)
        Displaced_Lilian = Current_Lilian - NumberOfDays[i];
    else
        Displaced_Lilian = Current_Lilian + NumberOfDays[i];

    CEEDATE ( &Displaced_Lilian, &picstr, chrdate, &FC );
    if ( _FBCHECK (FC , CEE000) == 0 )
    {
        chrdate[8] = '\0';
    }
}

Compare the dates to an array of expected values. Issue an error message if any conversion is incorrect.

if ( memcmp ( &chrdate, ChkDates[i] , 8) != 0)
printf ("Error in returned date %8s for displacement %d
", chrdate,NumberOfDays[i]);

Call CEEDYWK to return the day-of-the-week value (1 thru 7) for each calculated Lilian date. Compare results to an array of expected returned values and issue an error message for any incorrect values.

CEEDYWK ( &Displaced_Lilian, &WeekDay, &FC );
if ( _FBCHECK (FC , CEE000) == 0 )
{
    if ( WeekDay != ChkWeekDay[i])
        printf ( "Error in day of the week for %s
", chrdate);
    else
        printf ("Error finding day of the week\n"");
    CEEMSG(&FC, &dest, NULL);
}
else
{
    printf ("Error converting Lilian date to date.\n" );
    CEEMSG(&FC, &dest, NULL);
}
} /* for loop */
Calls to CEEDAYS, CEEDATE, and CEEDYWK in COBOL

Calls to CEEDAYS, CEEDATE, and CEEDYWK in COBOL

**CEEDAYS** - convert date to Lilian format
**CEEDATE** - convert Lilian date to date
**CEEDYWK** - find day of week from Lilian

CEEDAYS is passed the calendar date "11/09/92". The date is originally in YYMMDD format and conversion to Lilian format takes place. On return from CEEDAYS, a varying number of days is added to or subtracted from the Lilian date. CEEDATE is then called to convert the Lilian dates to the format "MM/DD/YY". CEEDYWK is called to return the day of the week for each derived Lilian date.

The results are tested for accuracy.

IDENTIFICATION DIVISION.
PROGRAM-ID. CE77DAT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WEEKDAY PIC S9(9) BINARY.
01 LILIAN PIC S9(9) BINARY.
01 CURRENT-LILIAN PIC S9(9) BINARY.
01 DISPLACED-LILIAN PIC S9(9) BINARY.
01 FC COPY CEEIGZCT.
02 Condition-Token-Value.
  03 Case-1-Condition-ID.
    04 Severity PIC S9(4) BINARY.
    04 Msg-No PIC S9(4) BINARY.
  03 Case-2-Condition-ID REDEFINES Case-1-Condition-ID.
    04 Class-Code PIC S9(4) BINARY.
    04 Cause-Code PIC S9(4) BINARY.
  03 Case-Sev-Ctl PIC X.
  03 Facility-ID PIC XXX.
02 I-S-Info PIC S9(9) BINARY.
01 INDXX PIC S9(9) BINARY.
01 NUMBER-OF-DAYS.
  05 NUMBERS.
    10 FILLER PIC S9(9) BINARY VALUE 6.
    10 FILLER PIC S9(9) BINARY VALUE 4.
  05 NUMBEROFDAYS REDEFINES NUMBERS PIC S9(9) BINARY OCCURS 5 TIMES.
01 PICSTR.
  02 Vstring-length PIC S9(4) BINARY.
  02 Vstring-text.
  03 Vstring-char PIC X, OCCURS 0 TO 256 TIMES DEPENDING ON Vstring-length of PICSTR.
01 CHRDATE PIC X(80).
01 CURRENT-DATE.
  02 Vstring-length PIC S9(4) BINARY.
  02 Vstring-text.
    03 Vstring-char PIC X, OCCURS 0 TO 256 TIMES DEPENDING ON Vstring-length of CURRENT-DATE.
01 INPUT-VARIABLES.
  05 DATE-TABLE.
    10 FILLER PIC X(9) VALUE "08/21/92".
    10 FILLER PIC X(9) VALUE "11/29/92".
    10 FILLER PIC X(9) VALUE "11/19/92".
    10 FILLER PIC X(9) VALUE "11/04/92".
    10 FILLER PIC X(9) VALUE "11/13/92".
  05 CHKDATES REDEFINES DATE-TABLE PIC X(9) OCCURS 5 TIMES.
01 CHK-WEEKDAYS.
  05 DAY-TABLE.
    10 FILLER PIC S9(9) BINARY VALUE 6.
PROCEDURE DIVISION.
       0001-BEGIN-PROCESSING.
       DISPLAY "*** Example CE77DAT in motion".*    *********************************************************
       * The current date is converted to a Lilian date.       *
       * *********************************************************
       MOVE  6 TO Vstring-length of PICSTR.
       MOVE "MM/DD/YY" TO Vstring-text of PICSTR.
       * *********************************************************
       * Call CEEDAYS to return the Lilian days tally for the *
       * date value in the variable CURRENT-DATE.              *
       * *********************************************************
       CALL "CEEDAYS" USING CURRENT-DATE, PICSTR,
       CURRENT-LILIAN, FC.
       IF NOT CEE000 THEN
           DISPLAY "Error " Msg-No of FC
           " in converting current date"
       END-IF.
       * *********************************************************
       * The datestamp mask must be changed for the dates *
       * being entered by the user.                            *
       * *********************************************************
       MOVE  8 TO Vstring-length of PICSTR.
       MOVE "YYMMDD" TO Vstring-text of PICSTR.
       * ***********************************************************
       * In the following loop, add or subtract the number of *
       * Lilian dates. Determine the day of the week for each *
       * Lilian date and convert each date back to "MM/DD/YY" *
       * format. Issue a message if anything goes wrong.         *
       * ***********************************************************
       MOVE 1 TO INDXX.
       PERFORM UNTIL INDXX = 6
           IF (INDXX = 1 OR 4) THEN
               COMPUTE DISPLACED-LILIAN =
               CURRENT-LILIAN - NUMBEROFDAYS(INDXX)
           ELSE
               COMPUTE DISPLACED-LILIAN =
               CURRENT-LILIAN + NUMBEROFDAYS(INDXX)
           END-IF
           * Call CEEDATE to convert the Lilian dates to *
           * "MM/DD/YY" format.
           * ***********************************************************
           CALL "CEEDATE" USING DISPLACED-LILIAN, PICSTR,
           CHRDATE, FC
           IF CEE000 THEN
               ********** ****************************************************
               * Compare converted date to expected value *
               ********** ****************************************************
               IF CHRDATE NOT = CHKDATES(INDXX) THEN
                   DISPLAY "Expecting returned date of "
                   CHKDATES(INDXX)
                   " for displacement of " NUMBEROFDAYS(INDXX)
                   ", but got returned date of " CHRDATE
               END-IF
               ********** ****************************************************
               * Call CEEDYWK to return a day-of-the week value (1 *
               * thru 7) for each calculated Lilian date. Compare *
               * results to an array of expected values and issue *
               * an error message for any incorrect values.          *
               * ***********************************************************
               CALL "CEEDYWK" USING DISPLACED-LILIAN, WEEKDAY, FC
               IF CEE000 THEN
                   IF WEEKDAY NOT = CHKWEEKDAY(INDXX) THEN
                       DISPLAY "Expecting day of week "
                       CHKWEEKDAY(INDXX) " , but got " WEEKDAY
                       " instead for " CHRDATE
                   END-IF
               ELSE
                   DISPLAY "Error " Msg-No of FC
                   " in finding day-of-week"
Calls to CEEDAYS, CEEDATE, and CEEDYWK in PL/I

*PROCESS MACRO;
* *FUNCTION : CEEDAYS - convert date to Lilian date
* : CEEDATE - convert Lilian Date to date
* : CEEDYWK - find day-of-week from Lilian
* *
* CEEDAYS is passed the calander date "11/09/92". The *
* date is originally in YYMMDD format and conversion to *
* Lilian format takes place. On return, a varying number *
* of days is added to or subtracted from the Lilian date. *
* CEEDATE is called to convert the Lilian dates to the *
* calendar format "MM/DD/YY". CEEDYWK is called to *
* return the day of the week for each derived Lilian date. *
* *
* The results are tested for accuracy.
*
*****************************************************************************************/
CE77DAT: PROC OPTIONS(MAIN);

%INCLUDE  CEEIBMAW;
%INCLUDE  CEEIBMCT;

DCL CHRDATE          CHAR(80);
DCL CURRENT_DATE     CHAR(255) VARYING;
DCL PICSTR           CHAR(255) VARYING;
DCL Lilian           REAL FIXED BINARY(31,0);
DCL ii               REAL FIXED BINARY(31,0);
DCL NumberOfDays (5) REAL FIXED BINARY(31,0)
  INIT(  80, 20, 10, 5, 4);
DCL ChkWeekDay   (5) REAL FIXED BINARY(31,0)
  INIT(  6,  1,  5, 4, 6);
DCL CURRENT_LILIAN   REAL FIXED BINARY(31,0);
DCL DISPLACED_LILIAN REAL FIXED BINARY(31,0);
DCL WEEKDAY          REAL FIXED BINARY(31,0);
DCL 01 FC,                      /* Feedback token */
          03 MsgSev    REAL FIXED BINARY(15,0),
          03 MsgNo     REAL FIXED BINARY(15,0),
          03 Flags,    /* Facility ID */
            05 Case      BIT(2),
            05 Severity  BIT(3),
            05 Control   BIT(3),
          03 FacID     CHAR(3),    /* Facility ID */
          03 ISI                   /* Instance-Specific Information */
          REAL FIXED BINARY(31,0);
DCL ChkDates (5)     CHAR(8) INIT(
'08/21/92',
'11/29/92',
'11/19/92',
'11/04/92',
'11/13/92');

PUT SKIP LIST( '*** Example CE77DAT complete');
STOP RUN.
Calls to CEECBLDY in COBOL

This example shows converting a 2-digit input date to a COBOL Integer date, adding 90 days to the Integer date, and converting the Integer format date back to a 4-digit year format using COBOL intrinsic functions.

CBL QUOTE
*************************************************
** Module/File Name: CBLDAYS
*********************************************************/
** Function: Invoke CEECBLDY callable service **
** to convert date to COBOL Lilian format. **
** This service is used when using the **
** Language Environment Century Window **
** mixed with COBOL Intrinsic Functions. **
*********************************************************/
IDENTIFICATION DIVISION.
PROGRAM-ID. CBLDAYS.

/* The date picstr must be adjusted to fit the current */
/* date format. */
Picstr = 'MM/DD/YY';

/***********************************************************/
/* In the following loop, add or subtract the number */
/* of days in each element of the NumberOfDays array to the */
/* Lilian date. Determine the day of the week for each */
/* Lilian date and convert each date back to "MM/DD/YY" */
/* format. Issue a message if anything goes wrong. */
/***********************************************************/
DO ii = 1 TO 5;
   IF ( ii= 1 | ii= 4 )  THEN  DO;
      Displaced_Lilian = Current_Lilian - NumberOfDays(ii); END;
   ELSE  DO;
      Displaced_Lilian = Current_Lilian + NumberOfDays(ii);
   END;
   CALL CEEDATE ( Displaced_Lilian, Picstr, ChrDate, FC );
   IF FBCHECK( FC, CEE000)  THEN  DO;
      PUT SKIP LIST( 'Error ' || FC.MsgNo || ' converting Date to Lilian Date' );
      END;
      ELSE  DO;
         PUT SKIP EDIT( 'Error in returned date ', Chrdate, ' for number of days ', NumberOfDays(ii) )
               ( (3) a, f(6) );
         END;
      END;
   ELSE  DO;
      PUT SKIP LIST( '<Error ' || FC.MsgNo || ' converting Date to Lilian Date' );
      END;
   END;
ELSE  DO;
   PUT SKIP LIST( 'Error ' || FC.MsgNo || ' converting Date to Lilian Date' );
   END;
   CALL CEEDYWK ( Displaced_Lilian, WeekDay, FC );
   IF FBCHECK( FC, CEE000)  THEN  DO;
      PUT SKIP EDIT( 'Error in day of the week for ', ChrDate)
                   ( a, a );
      END;
      ELSE  DO;
         PUT SKIP LIST( '<Error finding Day-of-Week');
         END;
         END;
   PUT SKIP LIST( '<<< Example CE77DAT complete');
END CE77DAT;
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CHRDATE.
   05 CHRDATE-LENGTH      PIC  9(2) BINARY.
   05 CHRDATE-STRING      PIC X(50).
01 PICSTR.
   05 PICSTR-LENGTH       PIC S9(4) BINARY.
   05 PICSTR-STRING       PIC X(50).
01 COBINT                  PIC S9(9) BINARY.
01 NEWDATE                 PIC 9(8).
01 FC                     PIC X(12).
PROCEDURE DIVISION.

*************************************************
** Specify input date and length               **
*************************************************
MOVE "1 January 00" to CHRDATE-STRING.
MOVE 25 TO CHRDATE-LENGTH.

*************************************************
** Specify a picture string that describes     **
** input date, and the picture string"s length.**
*************************************************
MOVE "ZD Mmmmmmmmmmmmmmmz YY" TO PICSTR-STRING.
MOVE 23 TO PICSTR-LENGTH.

*************************************************
** Call CEECBLDY to convert input date to a    **
** COBOL integer date                          **
*************************************************
CALL "CEECBLDY" USING CHRDATE, PICSTR, COBINT, FC.

*************************************************
** If CEECBLDY runs successfully, then compute **
** the date of the 90th day after the          **
** input date using Intrinsic Functions        **
*************************************************
IF (FC = LOW-VALUE) THEN
   COMPUTE COBINT = COBINT + 90
   COMPUTE NEWDATE = FUNCTION
   DATE-OF-INTEGER (COBINT)
   DISPLAY NEWDATE " is COBOL integer day: " COBINT
ELSE
   CONTINUE
END-IF.
GOBACK.
Date and time services
Chapter 21. National language support

This topic introduces the national language support services, which you use to set the national language, the country code, currency symbols, and decimal separators. It includes examples showing you how to query the default country code and change it, how to get the default date and time in the new country code, and how to convert the seconds to a timestamp. It also provides guidance for setting national language and country codes, including examples that show how national language services work in conjunction with date and time services.

Customizing Language Environment output for a given country

National language support services allow you to customize Language Environment output (such as messages, RPTOPTS reports, RPTSTG reports, or dumps) for a given country by specifying the following:

- The language in which runtime messages, days of the week, and months are displayed and printed
- A country code that indicates the default date and time format, currency symbol, decimal separator, and thousands separator

Related options:
COUNTRY
  Sets default country
NATLANG
  Sets initial national language

Related callable services:
CEE3CTY
  Sets default country
CEE3LNG
  Sets national language
CEE3MC2
  Gets default and international currency symbols
CEE3MCS
  Gets default currency symbol
CEE3MDS
  Gets default decimal separator
CEE3MTS
  Gets default thousands separator
CEEFMDA
  Gets default date format
CEEFMDT
  Gets default date and time format
CEEFMTM
  Gets default time format

Setting the national language

You can set the national language with the NATLANG runtime option or the CEE3LNG callable service. CEE3LNG is not supported in PL/I multitasking applications. The national language settings affect the
error messages, month name, and day of the week name. Message translations are provided for the following languages:

**ENU**  
Mixed-case U.S. English

**UEN**  
Uppercase U.S. English

**JPN**  
Japanese

### Setting the country code

You can use the COUNTRY runtime option or the CEE3CTY callable service to set the current country code for your application. The country code determines the default formats used to display and print the date and timestamps in the reports generated by the RPTSTG runtime option, RPTOPTS runtime option, and the CEE3DMP (dump) callable service. Default values associated with the country code also describe the currency symbol, decimal separator, and thousands separator.

Because CEE3LNG and CEE3CTY allow you to maintain multiple national languages and country settings on separate LIFO stacks, you can easily reset the national language or alternate between different country settings. For example, if you want to ensure that a routine in your application outputs the date and time in a Japanese format, use CEE3CTY to query the current default setting and, if necessary, to set it to Japanese with CEE3CTY if some other country code is in effect. For sample user code, see z/OS Language Environment Programming Reference.

The C/C++ language provides locales, which are UNIX structures that reflect different linguistic, cultural, and territorial conventions. Locale-sensitive C language functions make use of values and formats in the currently loaded locale. The Locale callable services exploit a subset of these C library interfaces for internationalized applications. See Chapter 22, “Locale callable services,” on page 319 for more information.

However, although the National Language Support callable services have some functional overlap with the Locale callable services, the two sets of services are completely independent of each other. Locale settings and the COUNTRY runtime option do not affect each other. Likewise, the Locale callable services and the National Language Support callable services are mutually exclusive. The National Language Support callable services derive values and formats only from defaults established by the COUNTRY runtime option or the CEE3CNTY service.

Language Environment does not currently support certain languages as national languages, so you would not be able to use CEE3LNG to set the national language to an unsupported language. You can, however, change the date and time format so that your English or Japanese banking application, for example, would display the default date and time format for an unsupported language. In general, you must use CEE3CTY to set the conventions for formatting date and time information.

### Euro support

The current country code determines the default currency symbol that will be returned by the CEE3MCS callable service.

For countries in the European Union that have adopted the Euro as the legal tender, the currency symbol is represented as a hex string in the default country settings. For specific values, see z/OS Language Environment Programming Reference. The value is taken from a typical code page for the given country, but, of course, the actual graphical representation depends on the code page in use.

Language Environment supports the Euro as the default currency symbol in the following countries: Austria, Belgium, Finland, France, Germany, Greece, Ireland, Italy, Luxembourg, the Netherlands, Portugal, and Spain. As more countries pass the Economic and Monetary Union convergence criteria and
adopt the Euro as the legal currency, the Euro sign will replace the national currency symbol as the default.

Combining national language support and date and time services

To customize your applications for a particular country, use national language support services to query the current country code, which you then can use as input to the Language Environment date and time callable services. For example, you could query the current country code with CEE3CTY and then use the returned value and CEEFMDT to get the default date and time format. When calling the CEEDATM (convert seconds to character timestamp) date and time service, you can use the string returned by CEEFMDT to specify the format of the convert seconds to character timestamp.

Calls to CEE3CTY, CEEFMDT, and CEEDATM in C

This example illustrates how you would query the default country code (CEE3CTY), change it to another country code (CEE3CTY), get the default date and time in the new country code (CEEFMDT), and convert the seconds to a timestamp (CEEDATM).

/*Module/File Name: EDCNLS */
/**********************************************************************/
/* FUNCTION */
/*             CEE3CTY   : query default country. set country to      */
/*                       : Germany.                                   */
/*             CEEFMDT   : get the German date and time format        */
/*             CEEDATM   : convert seconds to timestamp               */
/*                                                                    */
/* This example shows how to use several of the LE national           */
/* language support callable services. The current country is queried */
/* and changed to Germany. The default date and time for Germany is */
/* obtained. CEEFMDT is called to convert a large numeric value in */
/* seconds to the timestamp 16.05.1988 19:01:01. */
/**********************************************************************/
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <stdlib.h>
#include <ceedcct.h>

int main(void) {
    _FEEDBACK fc;
    _INT4 function;
    _CHAR2 country, symbol;
    _CHAR80 date_pic;
    _FLOAT8 seconds;
    _VSTRING  picstr;
    _CHAR80 timestp;
    #define DE  "DE"
    #define BL  "  ">
    printf ( "\n**********************************\n");
    printf ( "CESCNLS C Example is now in motion\n");
    printf ( "\n**********************************\n");
    /**************************************************************/
    /* Call CEE3CTY to query the current country setting */
    /**************************************************************/
    function = 2;
    CEE3CTY( &function, country, &fc);
    if ( (_FBCHECK (fc , CEE000)) != 0 ) {
        printf("CEE3CTY failed with message number %d\n",fc.tok_msgno);
        exit(2999);
    }
    /**************************************************************/
    /* Call CEE3CTY to set current country to Germany. */
    /**************************************************************/
    function = 3;
    CEE3CTY( &function, DE, &fc);
    if ( (_FBCHECK (fc , CEE000)) != 0 ) {
        printf("CEE3CTY failed with message number %d\n",fc.tok_msgno);
        exit(2999);
    }

Calls to CEE3CTY, CEEFMDT, and CEEDATM in COBOL

The following example illustrates how you would query the default country code (CEE3CTY), change it to another country code (CEE3CTY), get the default date and time in the new country code (CEEFMDT), and convert the seconds to a timestamp (CEEDATM).

```c
/* Call CEEFMDT retrieve the default date and time format */
CEEFMDT(BL,date_pic,&fc);
if ( (_FBCHECK (fc , CEE000)) != 0 ) {
    printf("CEEEDTNN failed with message number %d\n",fc.tok_msgno);
    exit(2999);
}
/* Call CEEDATM to convert the number of seconds from 12:00AM */
/* October 14, 1582 to 7:01PM May 16, 1988 to character */
/* the default country, Germany. */
seconds = 12799191661.986;
strcpy(picstr.string,date_pic);
picstr.length = strlen(picstr.string);
CEEDATM ( &seconds , &picstr , timestp , &fc );
if ( (_FBCHECK (fc , CEE000)) != 0 ) {
    printf("CEE3MDS failed with message number %d\n",fc.tok_msgno);
    exit(2999);
}
printf("Generated timestamp: %s",timestp);
printf ("\n*********************\n");
printf ("CESCNLS example ended.");
printf ("\n*********************\n");
```
REDEFINES Case-1-Condition-ID.
04 Class-Code PIC S9(4) BINARY.
04 Cause-Code PIC S9(4) BINARY.
 03 Case-Sev-Ctl PIC X.
 03 Facility-ID PIC XXX.
 02 I-S-Info PIC S9(9) BINARY.
 01 QUERY-COUNTRY-SETTING PIC S9(9) BINARY VALUE 2.
 01 SET-COUNTRY-SETTING PIC S9(9) BINARY VALUE 3.
 01 COUNTRY-PIC-STR PIC X(80).

PROCEDURE DIVISION.
 0001-BEGIN-PROCESSING.
    DISPLAY "**************************************".
    DISPLAY "CESCNLS COBOL example is now in motion.".
    DISPLAY "**************************************".

    ************************************************************
    *      Query Country Setting                              *
    ************************************************************
    MOVE QUERY-COUNTRY-SETTING TO FUNCTN.
    CALL "CEE3CTY" USING FUNCTN, COUNTRY, FC.
    IF NOT CEE000 of FC THEN
      DISPLAY "Error " Msg-No of FC
      " in query of country setting"
    ELSE
      ************************************************************
      *        Call CEE3CTY to set country to Germany           *
      ************************************************************
      MOVE SET-COUNTRY-SETTING TO FUNCTN
      MOVE GERMANY TO COUNTRY
      CALL "CEE3CTY" USING FUNCTN, COUNTRY, FC
      IF NOT CEE000 of FC THEN
        DISPLAY "Error " Msg-No of FC
        " in setting country"
      ELSE
        ************************************************************
        *            Call CEEDATM to convert the number of      *
        *            seconds from October 14, 1582 12:00AM       *
        *            to 16 May 1988 7:01PM to character format.  *
        *            that of the default country, Germany.       *
        ************************************************************
        MOVE 12799191661.986 TO SECONDS
        COMPUTE Vstring-length OF PICSTR =
           FUNCTION MIN( LENGTH OF COUNTRY-PIC-STR, 256 )
        MOVE COUNTRY-PIC-STR TO Vstring-text of PICSTR
        CALL "CEEDATM" USING SECONDS, PICSTR, TIMESTP, FC
        IF CEE000 of FC THEN
          DISPLAY "Generated timestamp is: " TIMESTP
          ELSE
            DISPLAY "Error " Msg-No of FC
            " generating timestamp"
        END-IF
      END-IF
      END-IF.
    END-IF.
    GOBACK.
Example using CEE3CTY, CEEFMDT, and CEEDATM in PL/I

Following is an example of querying and setting the country code and getting the date and time format in PL/I.

*PROCESS MACRO;
/*Module/File Name: IBMNLS
/****************************************************/
/*                                                        */
/* Function      CEE3CTY   : query default country        */
/*               CEEFMDT   : obtain the default date and  */
/*                           time format                  */
/*               CEEDATM   : convert seconds to timestamp */
/*                                                        */
/* This example shows how to use several of the LE      */
/* national language support callable services in a     */
/* PL/I program. The current country is queried, saved, */
/* and then changed to Germany. The default date and    */
/* time for Germany is obtained. CEEDATM is called to    */
/* convert a large numeric value in seconds to the      */
/* timestamp 16.05.1988 19:01:01 (May 16, 1988 7:01PM). */
/*                                                        */
/************************************************************/
CESCNLS: PROC OPTIONS(MAIN);
%INCLUDE  CEEIBMAW;
%INCLUDE  CEEIBMCT;
DCL FUNCTN          REAL FIXED BINARY(31,0);
DCL QUERY_COUNTRY   REAL FIXED BINARY(31,0) INIT(2);
DCL SET_COUNTRY     REAL FIXED BINARY(31,0) INIT(3);
DCL SECONDS         REAL FLOAT DECIMAL(16);
DCL COUNTRY         CHARACTER ( 2 );
DCL GERMANY         CHARACTER ( 2 )INIT ('DE');
DCL 01 FC,                      /* Feedback token */
03 MsgSev    REAL FIXED BINARY(15,0),
03 MsgNo     REAL FIXED BINARY(15,0),
03 Flags,        03 Case      BIT(2),
05 Severity  BIT(3),
05 Control   BIT(3),
03 FacID     CHAR(3),    /* Facility ID */
03 ISI                   /* Instance-Specific Information */
REAL FIXED BINARY(31,0);
DCL TIMESTP         CHAR(80);
DCL PICSTR          CHAR(80);
DCL PIC_VSTR        CHAR(255) VARYING;
/**********************************************************/
/* Query country setting                                *//**********************************************************/
FUNCTN = QUERY_COUNTRY;
CALL CEE3CTY ( FUNCTN, COUNTRY, FC );
IF  FBCHECK( FC, CEE000)  THEN  DO;
        /****************************************************/ /* Call CEE3CTY to set country to Germany */
        /****************************************************/
FUNCTN = SET_COUNTRY;
COUNTRY = GERMANY;
CALL CEE3CTY ( FUNCTN, COUNTRY, FC );
IF ^ FBCHECK( FC, CEE000) THEN DO;
        PUT SKIP LIST('Error ' || FC.MsgNo || ' in setting country');
        END;
ELSE DO;
        /****************************************************/ /* Call CEEFMDT to get default date/time format for */
        /* Germany and verify format against published value. */
        /****************************************************/
COUNTRY = ' '
CALL CEEFMDT ( COUNTRY, PICSTR, FC );
IF ^ FBCHECK( FC, CEE000) THEN DO;
        PUT SKIP LIST( 'Error ' || FC.MsgNo
       | ' getting default date/time format for Germany.');
        END;
ELSE DO;
        /****************************************************/ /* Call CEEDATM to convert the number representing */
        /* the number of seconds from October 14, 1582 */
        /****************************************************/
        CEEDATM = COUNTRY;
        CALL CEEDATM ( COUNTRY, SECONDS, FC );
        CALL CEE3CTY ( FUNCTN, COUNTRY, FC );
        IF  FBCHECK( FC, CEE000)  THEN  DO;
        IF ^ FBCHECK( FC, CEE000) THEN DO;
        PUT SKIP LIST( 'Error ' || FC.MsgNo
       | ' in obtaining default date and time format.');
        END;
        ELSE DO;
        /****************************************************/ /* Call CEEDATM to convert the number representing */
        /* the number of seconds from October 14, 1582 */
        /****************************************************/
/ matches that of the default country, Germany. */
/**********************************************************************/
SECONDS = 12799191661.986;
PIC_VSTR = PICSTR;
CALL CEEDATM ( SECONDS, PIC_VSTR, TIMESTP, FC );
IF FBCHECK( FC, CEE000) THEN DO;
  PUT SKIP EDIT ('Generated timestamp is ',
               TIMESTP) (A, A);
  END;
ELSE DO;
  PUT SKIP LIST ( 'Error ' || FC.MsgNo
                  || ' generating timestamp');
  END;
END;
END;
ELSE DO;
  PUT SKIP LIST ( 'Error ' || FC.MsgNo || ' querying country code');
  END;
END CESCNLS;
Chapter 22. Locale callable services

This topic describes how to use the Language Environment locale callable services to internationalize your applications, and includes examples that show how locale callable services work in conjunction with each other. Locale callable services do not affect, nor are they affected by, Language Environment callable services or the COUNTRY or NATLANG runtime options. Language Environment locale callable services are not supported in PL/I multitasking applications.

Language Environment locale support adheres to the standards used by C. For detailed information about these standards, locales, and charmaps, see z/OS XL C/C++ Programming Guide.

Customizing Language Environment locale callable services

Locale callable services allow you to customize culturally-sensitive output for a given national language, country, and code set by specifying a locale name.

Related callable services:

- **CEEFMON**
  - Formats monetary string
- **CEEFTDS**
  - Formats date and time into a character string
- **CEELCNV**
  - Query locale numeric conventions
- **CEEQDTC**
  - Queries locale, date, and time conventions
- **CEEQRYL**
  - Queries the active locale environment
- **CEESCOL**
  - Compares the collation weights of two strings
- **CEESETL**
  - Sets the locale operating environment
- **CEESTXF**
  - Transforms string characters into collation weights

See z/OS Language Environment Programming Reference for syntax information.

Although C or C++ routines can use the locale callable services, it is recommended that they use the equivalent native C library services instead for portability across platforms. Table 56 on page 319 shows the Language Environment locale callable services and the equivalent C library routines.

<table>
<thead>
<tr>
<th>Language Environment locale callable service</th>
<th>C library routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEFMON</td>
<td>strftime()</td>
</tr>
<tr>
<td>CEEFTDS</td>
<td>strptime()</td>
</tr>
<tr>
<td>CEELCNV</td>
<td>localeconv()</td>
</tr>
<tr>
<td>CEEQDTC</td>
<td>localdtconv()</td>
</tr>
<tr>
<td>CEEQRYL</td>
<td>setlocale()</td>
</tr>
</tbody>
</table>
Developing internationalized applications

Locale callable services define environment control variables that you can set to establish language-specific information and preferences for an application. Locale callable services also provide a means for establishing global preferences, such as setlocale(), locale management services, and locale-dependent interfaces to the application.

Locale callable services allow you to develop applications that can be used in multiple countries, because they can function with specific language and cultural conventions. Such applications are referred to as internationalized applications. These applications have no built-in assumptions with respect to the language, culture, or conventions of their users or the data they process. Instead, language and cultural information is set at run time, a process called localization. Thus, the application processes data provided specifically for a certain locale. In Language Environment, localization occurs at the enclave level.

Examples of using locale callable services

The following topics demonstrate how to use locale callable services in your applications.

Example calls to CEEFMON

The following examples illustrate calls to CEEFMON to convert a numeric value to a monetary string using a specified format.

Calls to CEEFMON in COBOL

```cobol
CBL LIB,QUOTE
*Module/File Name: IGZTFMON
*************************************************
* Example for callable service CEEFMON         *
* Function: Convert a numeric value to a      *
* monetary string using specified           *
* format passed as parameter.               *
* Valid only for COBOL for MVS & VM Release 2 *
* or later.                                   *
*************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. COBFMON.
DATA DIVISION. WORKING-STORAGE SECTION.
  01 Monetary      COMP-2.
  01 Max-Size     PIC S9(9) BINARY.
  01 Format-Mon.
    02 FM-Length  PIC S9(4) BINARY.
    02 FM-String  PIC X(256).
  01 Output-Mon.
    02 OM-Length  PIC S9(4) BINARY.
    02 OM-String  PIC X(60).
  01 Length-Mon  PIC S9(9) BINARY.
  01 Locale-Name.
    02 LN-Length  PIC S9(4) BINARY.
    02 LN-String  PIC X(256).
** Use Locale category constants
COPY CEEIGZLC.
  01 FC.
    02 Condition-Token-Value.
    COPY CEEIGZCT.
      03 Case-1-Condition-ID.
        04 Severity  PIC S9(4) BINARY.
```
**PROCEDURE DIVISION.**

**Set up locale name for United States**
MOVE 14 TO LN-Length.
MOVE "En_US.IBM-1047" TO LN-String (1:LN-Length).
**Set all locale categories to United States.**
**Use LC-ALL category constant from CEEIGZLC.**
CALL "CEESETL" USING Locale-Name, LC-ALL, FC.
**Check feedback code**
IF Severity > 0
  DISPLAY "Call to CEESETL failed. " Msg-No
  STOP RUN
END-IF.
**Set up numeric value**
MOVE 12345.62 TO Monetary.
MOVE 60 TO Max-Size.
MOVE 2 TO FM-Length.
MOVE "%i" TO FM-String (1:FM-Length).
**Call CEEFMON to convert numeric value**
CALL "CEEFMON" USING OMITTED, Monetary, Max-Size, Format-Mon,
  Output-Mon, Length-Mon, FC.
**Check feedback code and display result**
IF Severity > 0
  DISPLAY "Call to CEEFMON failed. " Msg-No
ELSE
  DISPLAY "International format is "
  OM-String(1:OM-Length)
END-IF.
STOP RUN.
END PROGRAM COBFMON.

Calls to CEEFMON in PL/I

*PROCESS MACRO;
/*Module/File Name: IBMFMON */
/*****************************/
/* Example for callable service CEEFMON */
/*****************************/
PLIFMON: PROC OPTIONS(MAIN);
%INCLUDE CEEIBMAW; /* ENTRY defs, macro defs */
%INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants */
%INCLUDE CEEIBMLC; /* Locale category constants */
/* CEESETL service call arguments */
DCL LOCALE_NAME CHAR(14) VARYING;
/* CEEFMON service call arguments */
DCL MONETARY REAL FLOAT DEC(16); /* input value */
DCL MAXSIZE_FMON BIN FIXED(31); /* output size */
DCL FORMAT_FMON CHAR(256) VARYING; /* format spec */
DCL RESULT_FMON BIN FIXED(31); /* result status */
DCL OUTPUT_FMON CHAR(60) VARYING; /* output string */
DCL 01 FC, /* Feedback token */
  03 MsgSev REAL FIXED BINARY(15,0),
  03 MsgNo REAL FIXED BINARY(15,0),
  03 Flags,
    05 Case BIT(2),
    05 Severity BIT(3),
    05 Control BIT(3),
  03 FacID CHAR(3), /* Facility ID */
  03 ISI /* Instance-Specific Information */
    REAL FIXED BINARY(31,0);
/* init locale name to United States */
  LOCALE_NAME = 'En_US.IBM-1047';
/* use LC-ALL category constant from CEEIBMLC */
  CALL CEESETL (LOCAL_NAME, LC_ALL, FC);
/* FBCHECK macro used (defined in CEEIBMCT */
Example calls to CEEFTDS

The following examples illustrate calls to CEEFTDS to convert a numeric time and date to a string using a specified format.

Calls to CEEFTDS in COBOL

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. MAINFTDS.
DATA DIVISION.
WORKING-STORAGE SECTION.
* Use TD-Struct for CEEFTDS calls
COPY CEEIGZTD.
*
PROCEDURE DIVISION.
* Subroutine needed for pointer addressing
CALL "COBFTDS" USING TD-Struct.
STOP RUN.
* IDENTIFICATION DIVISION.
PROGRAM-ID. COBFTDS.
DATA DIVISION.
WORKING-STORAGE SECTION.
* UseLocale category constants
COPY CEEIGZLC.
*
01 01-PTR-FTDS POINTER.
01 Output-FTDS.
 02 0-Length PIC S9(4) BINARY.
 02 0-String PIC X(72). 01 Format-FTDS.
 02 F-Length PIC S9(4) BINARY.
 02 F-String PIC X(64).
 01 Max-Size PIC S9(9) BINARY.
 01 FC.
```
02 Condition-Token-Value.
COPY CEEIGZCT.

03 Case-1-Condition-ID.
  04 Severity PIC S9(4) BINARY.
  04 Msg-No PIC S9(4) BINARY.
03 Case-2-Condition-ID
REDEFINES Case-1-Condition-ID.
  04 Class-Code PIC S9(4) BINARY.
  04 Cause-Code PIC S9(4) BINARY.
03 Case-Sev-Ctl PIC X.
03 Facility-ID PIC XXX.

02 I-S-Info PIC S9(9) BINARY.

LINKAGE SECTION.
* Use TD-Struct for calls to CEEFTDS
COPY CEEIGZTD.
* PROCEDURE DIVISION USING TD-Struct.
* Set up time and date values
  MOVE 1 TO TM-Sec.
  MOVE 2 TO TM-Min.
  MOVE 3 TO TM-Hour.
  MOVE 9 TO TM-Day.
  MOVE 11 TO TM-Mon.
  MOVE 94 TO TM-Year.
  MOVE 5 TO TM-Wday.
  MOVE 344 TO TM-Yday.
  MOVE 1 TO TM-Is-DLST.
* Set up format string for CEEFTDS call
  MOVE 72 TO Max-Size.
  MOVE 36 TO F-Length.
  MOVE "Today is %A, %b %d  Time: %I:%M  %p" TO F-String (1:F-Length).
* Set up pointer to structure for CEEFTDS call
  SET Ptr-FTDS TO ADDRESS OF TD-Struct.
* Call CEEFTDS to convert numeric values
  CALL "CEEFTDS" USING OMITTED, Ptr-FTDS,
    Max-Size, Format-FTDS,
    Output-FTDS, FC.
* Check feedback code and display result
  IF Severity = 0
    DISPLAY "Format " F-String (1:F-Length)
    DISPLAY "Result " O-String (1:O-Length)
  ELSE
    DISPLAY "Call to CEEFTDS failed. " Msg-No
  END-IF.
EXIT PROGRAM.
END PROGRAM COBFTDS.
* END PROGRAM MAINFTDS.

Calls to CEEFTDS in PL/I

*PROCESS MACRO;
  /*Module/File Name: IBMFTDS */
  /**********************************************************************/
  /* Example for callable service CEEFTDS */
  /* Function: Convert numeric time and date values */
  /* to a string based on a format specification */
  /* string parameter and locale format conversions */
  /**********************************************************************/
PLIFTDS: PROC OPTIONS(MAIN);
  %INCLUDE CEEIBMAM; /* ENTRYdefs, macrodefs */
  %INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants */
  %INCLUDE CEEIBMLC; /* Locale category constants */
  %INCLUDE CEEIBMTD; /* TD_STRUCT for CEEFTDS calls */
  /* use explicit pointer to local TD_STRUCT structure*/
  DCL TIME_AND_DATE POINTER INIT(ADDR(TD_STRUCT)));
  /* CEEFTDS service call arguments */
  DCL MAXSIZE_FTDS BIN FIXED(31); /* OUTPUT_FTDS size */
  DCL FORMAT_FTDS CHAR(64) VARYING; /* format string */
  DCL OUTPUT_FTDS CHAR(72) VARYING; /* output string */
Example calls to CEELCNV and CEESETL

The following examples illustrate calls to CEELCNV to retrieve the numeric and monetary format for the default locale, and to CEESETL to set the locale. These examples also indicate how to access the ISO/IEC 9899:1999 (C99) compliant version of the NM-Struct (COBOL) or NM_Struct (PL/I) copy of the localeconv structure in the C library. For further details, see the discussion of the CEELCNV callable service in z/OS Language Environment Programming Reference.

Calls to CEELCNV and CEESETL in COBOL

```
CBL LIB,QUOTE
*Module/File Name: IGZTLCNV

************************************************************************************
** Example for callable service CEELCNV **
** Function: Retrieve numeric and monetary **
** format for default locale and **
** print an item. **
** Set locale to France, retrieve **
** structure, and print an item. **
** Valid only for COBOL for MVS & VM Release 2 **
** or later. **
************************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. MAINLCNV.
DATA DIVISION.
WORKING-STORAGE SECTION.
************************************************************************************
** Use Locale NM-Struct for CEELCNV calls **
************************************************************************************
COPY CEEIGZN2.
*
PROCEDURE DIVISION.
************************************************************************************
** Subroutine needed for addressing **
************************************************************************************
CALL "COBLCNV" USING NM-Struct.
```
STOP RUN.
*
IDENTIFICATION DIVISION.
PROGRAM-ID.  COBLCNV.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01  PTR1 Pointer.
      01  Locale-Name.
          02  LN-Length PIC S9(5) BINARY.
          02  LN-String PIC X(256).
      *************************************************
      ** Use Locale category constants           **
      *************************************************
      COPY CEEIGZLC.

  01  FC.
      02  Condition-Token-Value.
      COPY CEEIGZCT.
          03  Case-1-Condition-ID.
              04  Severity    PIC S9(4) BINARY.
              04  Msg-No      PIC S9(4) BINARY.
          03  Case-2-Condition-ID.
              04  Class-Code  PIC S9(4) BINARY.
              04  Cause-Code  PIC S9(4) BINARY.
          03  Case-Sev-Ctl    PIC X.
          03  Facility-ID     PIC XXX.
      02  I-S-Info            PIC S9(9) BINARY.

LINKAGE SECTION.
  *************************************************
  ** Use Locale NM-Struct for CEELCNV calls      **
  *************************************************
  COPY CEEIGZN2.

  *  PROCEDURE DIVISION USING NM-Struct.
  *************************************************
  ** Call CEELCNV to retrieve values for locale**
  *************************************************
  CALL "CEELCNV" USING OMITTED, ADDRESS OF NM-Struct, FC.

  *************************************************
  ** Check feedback code and display result      **
  *************************************************
  IF Severity = 0 THEN
      DISPLAY "Default decimal point is "
      DECIMAL-PT-String(1:DECIMAL-PT-Length)
  ELSE
      DISPLAY "Call to CEELCNV failed. " Msg-No
  END-IF.

  *************************************************
  ** Set up locale for France                   **
  *************************************************
  MOVE 5 TO LN-Length.
  MOVE "Fr_FR" TO LN-String (1:LN-Length).

  *************************************************
  ** Call CEESETL to set monetary locale        **
  *************************************************
  CALL "CEESETL" USING Locale-Name, LC-MONETARY, FC.

  *************************************************
  ** Call CEESETL to set numeric locale         **
  *************************************************
  CALL "CEESETL" USING Locale-Name, LC-NUMERIC, FC.

  *************************************************
  ** Check feedback code and call CEELCNV again **
  ** using version 2 to get at C99 mapping.**
  *************************************************
  IF Severity = 0
      MOVE 2 TO Version
      set PTR1 to address of Version-Info
      CALL "CEELCNV" USING PTR1, ADDRESS OF NM-Struct, FC
      IF Severity > 0
          DISPLAY "Call to CEELCNV failed. "
          Msg-No
      ELSE
DISPLAY "French decimal point is ",
DECIMAL-PT-String(1:DECIMAL-PT-Length)
END-IF
ELSE
DISPLAY "Call to CEESETL failed. " Msg-No
END-IF.
EXIT PROGRAM.
END PROGRAM COBLCNV.
*
END PROGRAM MAINLCNV.

Calls to CEELCNV and CEESETL in PL/I

*PROCESS MACRO;
/!*Module/File Name: IBMLCNV                         */
/!* Example for callable service CEELCNV */
/* Function: Retrieve numeric and monetary format */
/* structure for default locale and print an item. */
/* Set locale to France, retrieve structure and */
/* print an item. */
/!**************************************************************************/
Example calls to CEEQDTC and CEESETL

The following examples illustrate calls to CEEQDTC to retrieve the date and time conventions, and to CEESETL to set the locale. To see how to access ISO/IEC 9899:1999 (C99) extensions, refer to the CEEELCNV examples. For specific details regarding parameter and copy file usage, see the discussion of the CEEQDTC callable service in z/OS Language Environment Programming Reference.

Calls to CEEQDTC and CEESETL in COBOL

```cobol
CBL LIB,QUOTE
*Module/File Name: IGZTQDTC
******************************************************************************
* Example for callable service CEEQDTC *
* MAINQDTC - Retrieve date and time convention structures for two countries and *
* compare an item. *
* Valid only for COBOL for MVS & VM Release 2 *
* or later. *
******************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. MAINQDTC.
DATA DIVISION.
WORKING-STORAGE SECTION.
* Use DTCONV structure for CEEQDTC calls
COPY CEEIGZDT.
*
PROCEDURE DIVISION.
* Subroutine needed for addressing
CALL "COBQDTC" USING DTCONV.
STOP RUN.
*
IDENTIFICATION DIVISION.
PROGRAM-ID. COBQDTC.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Locale-Name.
  02 LN-Length PIC S9(4) BINARY.
  02 LN-String PIC X(256).
* Use Locale category constants
COPY CEEIGZLC.
*
01 Test-Length1 PIC S9(4) BINARY.
01 Test-String1 PIC X(80).
01 Test-Length2 PIC S9(4) BINARY.
01 Test-String2 PIC X(80).
01 FC.
  02 Condition-Token-Value.
COPY CEEIGZCT.
  03 Case-1-Condition-ID.
  04 Severity PIC S9(4) BINARY.
  04 Msg-No PIC S9(4) BINARY.
  03 Case-2-Condition-ID
REDEFINES Case-1-Condition-ID.
  04 Class-Code PIC S9(4) BINARY.
  04 Cause-Code PIC S9(4) BINARY.
  03 Case-Sev-Ctl PIC X.
  03 Facility-ID PIC XXX.
  02 I-S-Info PIC S9(9) BINARY.
*
LINKAGE SECTION.
* Use Locale structure DTCONV for CEEQDTC calls
COPY CEEIGZDT.
*
PROCEDURE DIVISION USING DTCONV.
* Set up locale for France
MOVE 4 TO LN-Length.
MOVE "FFEY" TO LN-String (1:LN-Length).
* Call CEESETL to set all locale categories
CALL "CEESETL" USING Locale-Name, LC-ALL, FC.
* Check feedback code
IF Severity > 0
  DISPLAY "Call to CEESETL failed. " Msg-No
END-IF.
* Call CEEQDTC for French values
```
CALL "CEEQDTC" USING OMITTED,
        ADDRESS OF DTCONV, FC.
* Check feedback code
  IF Severity > 0
    DISPLAY "Call to CEEQDTC failed. " Msg-No
    EXIT PROGRAM
  END-IF.
* Save date and time format for FFYEY locale
  MOVE D-T-FMT-Length IN DTCONV TO Test-Length1
  MOVE D-T-FMT-String IN DTCONV TO Test-String1
* Set up locale for French Canadian
  MOVE 4 TO LN-Length.
  MOVE "FCEY" TO LN-String (1:LN-Length).
* Call CEESETL to set locale for all categories
  CALL "CEESETL" USING Locale-Name, LC-ALL,
        FC.
* Check feedback code
  IF Severity > 0
    DISPLAY "Call to CEESETL failed. " Msg-No
    EXIT PROGRAM
  END-IF.
* Call CEEQDTC again for French Canadian values
  CALL "CEEQDTC" USING OMITTED,
        ADDRESS OF DTCONV, FC.
* Check feedback code and display results
  IF Severity = 0
    * Save date and time format for FCEY locale
    MOVE D-T-FMT-Length IN DTCONV TO Test-Length2
    MOVE D-T-FMT-String IN DTCONV TO Test-String2
    IF Test-String1(1:Test-Length1) =
       Test-String2(1:Test-Length2)
      DISPLAY "Same date and time format."
    ELSE
      DISPLAY "Different formats."
      DISPLAY Test-String1(1:Test-Length1)
      DISPLAY Test-String2(1:Test-Length2)
    END-IF
  ELSE
    DISPLAY "Call to CEEQDTC failed. " Msg-No
  END-IF.
  EXIT PROGRAM.
END PROGRAM COBQDTC.
* END PROGRAM MAINQDTC.

Calls to CEEQTDC and CEESETL in PL/I

*PROCESS MACRO;
/*Module/File Name: IBMQDTC                         */
/****************************************************/
/* Example for callable service CEEQTDC             */
/* Function: Retrieve date and time convention      */
/*  structures for two countries, compare an item.  */
/****************************************************/
PLICQDTC: PROC OPTIONS(MAIN);

%INCLUDE CEEEIBMMA; /* ENTRY defs, macro defs */
%INCLUDE CEEEIBMCT; /* FBCHECK macro, FB constants */
%INCLUDE CEEEIBMCLC; /* Locale category constants */
%INCLUDE CEEEIBMMDT; /* DTCONV for CEEQTDC calls */

/* use explicit pointer to local DTCONV structure */
DCL LOCALDT POINTER INIT(ADDR(DTCONV));

/* CEESETL service call arguments */
DCL LOCALE_NAME CHAR(256) VARYING;
DCL 1 DTCONVC LIKE DTCONV; /* Def Second Structure */
DCL 1 FC, /* Feedback token */
Example calls to CEESCOL

The following examples illustrate calls to CEESCOL to compare the collation of two character strings.

Calls to CEESCOL in COBOL

```
CBL LIB,QUOTE
*Module/File Name: IGZTSCOL
***********************************************************************
* Example for callable service CEESCOL       *
* COBSCOL - Compare two character strings   *
* and print the result.                     *
* Valid only for COBOL for MVS & VM Release 2 *
* or later.                                  *
***********************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. COBSCOL.

DATA DIVISION.
WORKING-STORAGE SECTION.
  01 String1.
     02 Strl-Length PIC 9(4) BINARY.
     02 Str1-String.
        03 Str1-Char PIC X OCCURS 0 TO 256 TIMES DEPENDING ON Str1-Length.
  01 String2.
     02 Str2-Length PIC 9(4) BINARY.
     02 Str2-String.
        03 Str2-Char PIC X OCCURS 0 TO 256 TIMES DEPENDING ON Str2-Length.
  01 Result PIC 9(9) BINARY.
  01 FC.
     02 Condition-Token-Value.
COPY CEEIGZCT.
     03 Case-1-Condition-ID.
     04 Severity PIC 9(4) BINARY.
     04 Msg-No PIC 9(4) BINARY.
     03 Case-2-Condition-ID
```
REDEFINES Case-1-Condition-ID.
04 Class-Code PIC S9(4) BINARY.
04 Cause-Code PIC S9(4) BINARY.
03 Case-Sev-Ctl PIC X.
03 Facility-ID PIC XXX.
02 I-S-Info PIC S9(9) BINARY.
*
PROCEDURE DIVISION.
******************************************************************************
* Set up two strings for comparison
******************************************************************************
MOVE 9 TO Str1-Length.
MOVE "12345a789" TO Str1-String (1:Str1-Length)
MOVE 9 TO Str2-Length.
MOVE "12346$789" TO Str2-String (1:Str2-Length)
******************************************************************************
* Call CEESCOL to compare the strings
******************************************************************************
CALL "CEESCOL" USING OMITTED, String1,
String2, Result, FC.
******************************************************************************
* Check feedback code
******************************************************************************
IF Severity > 0
  DISPLAY "Call to CEESCOL failed. " Msg-No
  STOP RUN
END-IF.
******************************************************************************
* Check result of compare
******************************************************************************
EVALUATE TRUE
  WHEN Result < 0
    DISPLAY "1st string < 2nd string."
  WHEN Result > 0
    DISPLAY "1st string > 2nd string."
  WHEN OTHER
    DISPLAY "Strings are identical."
END-EVALUATE.
STOP RUN.
END PROGRAM COBSCOL.

Calls to CEESCOL in PL/I

*PROCESS MACRO;
  /*Module/File Name: IBMSCOL */
  /* Example for callable service CEESCOL */
  /* Function: Compare two character strings and */
  /* print the result. */
  /*********************************************************************/

PLISCOL: PROC OPTIONS(MAIN);
%INCLUDE CEEIBMMAW; /* ENTRY defs, macro defs for LE */
%INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants */
%INCLUDE CEEIBMLC; /* Locale category constants */
/* CEESCOL service call arguments */
DCL STRING1 CHAR(256) VARYING;/* first string */
DCL STRING2 CHAR(256) VARYING;/* second string */
DCL RESULT_SCOL BIN FIXED(31);/* result of compare */
DCL 01 FC,                        /* Feedback token */
   03 MsgSev REAL FIXED BINARY(15,0),
   03 MsgNo REAL FIXED BINARY(15,0),
   03 Flags,
   05 Case BIT(2),
   05 Severity BIT(3),
   05 Control BIT(3),
   03 FacID CHAR(3),        /* Facility ID */
   03 ISI /* Instance-Specific Information */
      REAL FIXED BINARY(31,0);
STRING1 = '12345a789';
STRING2 = '12346$789';
CALL CEESCOL( *, STRING1, STRING2, RESULT_SCOL, FC);
Example calls to CEESETL and CEEQRYL

The following examples illustrate calls to CEESETL to set the locale, and to CEEQRYL to retrieve locale time information.

Calls to CEESETL and CEEQRYL in COBOL

```
CBL LIB,QUOTE
*Module/File Name: IGZTSETL
******************************************************************************
* Example for callable service CEESETL: set all locale categories to country  *
*   Sweden. Query one category.                                          *
******************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. COBSETL.
DATA DIVISION.
WORKING-STORAGE SECTION.
 01 Locale-Name.
    02 LN-Length PIC S9(4) BINARY.
    02 LN-String PIC X(256).
 01 Locale-Time.
    02 LT-Length PIC S9(4) BINARY.
    02 LT-String PIC X(256).
* Use Locale category constants
COPY CEEIGZLC.
* 01 FC.
    02 Condition-Token-Value.
    COPY CEEIGZCT.
      03 Case-1-Condition-ID.
        04 Severity PIC S9(4) BINARY.
        04 Msg-No PIC S9(4) BINARY.
      03 Case-2-Condition-ID
        REDEFINES Case-1-Condition-ID.
        04 Class-Code PIC S9(4) BINARY.
        04 Cause-Code PIC S9(4) BINARY.
      03 Case-Sev-Ctl PIC X.
      03 Facility-ID PIC XXX.
    02 I-S-Info PIC S9(9) BINARY.
* PROCEDURE DIVISION.
******************************************************************************
* Set up locale name for Sweden
******************************************************************************
MOVE 14 TO LN-Length.
MOVE 'Sv.SE.IBM-1047' TO LN-String (1:LN-Length).
******************************************************************************
* Set all locale categories to Sweden
* Use LC-ALL category constant from CEEIGZLC
******************************************************************************
CALL 'CEESETL' USING Locale-Name, LC-ALL, FC.
******************************************************************************
```

Locale callable services
* Check feedback code

*************************************************
IF Severity > 0
   DISPLAY 'Call to CEESETL failed. ' Msg-No
   STOP RUN
END-IF.

*************************************************
* Retrieve active locale for LC-TIME category
CALL 'CEEQRYL' USING LC-TIME, Locale-Time, FC.

*************************************************
* Check feedback code and correct locale
IF Severity = 0
   IF LT-String(1:LT-Length) =
      LN-String(1:LN-Length)
      DISPLAY 'Successful query.'
   ELSE
      DISPLAY 'Unsuccessful query.'
END-IF
ELSE
   DISPLAY 'Call to CEEQRYL failed. ' Msg-No
END-IF.
STOP RUN.
END PROGRAM COBSETL.

Calls to CEESETL and CEEQRYL in PL/I

**PROCESS MACRO;**
/* Module/File Name: IBMSETL */
/* Example for callable service CEESETL */
/* Function: Set all global locale environment */
/* categories to country. Query one category. */
/* */
/*****************************/
PLISETL: PROC OPTIONS(MAIN);
%INCLUDE CEEIBMAW; /* ENTRY deIs, macro deFs */
%INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants */
%INCLUDE CEEIBMLC; /* Locale category constants */
/* CEESETL service call arguments */
DCL LOCALE_NAME CHAR(14) VARYING;
/* CEEQRYL service call arguments */
DCL LOCALE_NAME_TIME CHAR(256) VARYING;
DCL 01 FC,                        /* Feedback token */
   03 MsgSev REAL FIXED BINARY(15,0),
   03 MsgNo REAL FIXED BINARY(15,0),
   03 Flags,
   05 Case BIT(2),
   05 Severity BIT(3),
   05 Control BIT(3),
   03 FacID CHAR(3),       /* Facility ID */
   03 ISI /* Instance-Specific Information */
      REAL FIXED BINARY(31,0);
/* init locale name with IBM default for Sweden */
LOCALE_NAME = 'Sv_SE.IBM-1047';
/* use LC_ALL category const from CEEIBMLC */
CALL CEESETL ( LOCALE_NAME, LC_ALL, FC );
/* FBCHECK macro used (defined in CEEIBMCT) */
IF FBCHECK( FC, CEE2KE ) THEN
   DO;  /* invalid locale name */
      DISPLAY ( 'Locale LC_ALL Call '||FC.MsgNo );
      STOP;
   END;
/* retrieve active locale for LC_TIME category */
/* use LC_TIME category const from CEEIBMLC */
CALL CEEQRYL ( LC_TIME, LOCALE_NAME_TIME, FC );
IF FBCHECK( FC, CEE000 ) THEN
   DO;  /* successful query, check category name */
      IF LOCALE_NAME_TIME ^= LOCALE_NAME THEN
         STOP;
      END;
   ELSE
      DISPLAY 'Call to CEEQRYL failed. ' Msg-No
   END-IF.
STOP RUN.
END PROGRAM COBSETL.
Example calls to CEEQRYL and CEESTXF

The following examples illustrate calls to CEEQRYL to retrieve the locale name, and to CEESTXF to translate a string into its collation weights.

Calls to CEEQRYL and CEESTXF in COBOL

```
CBL LIB,QUOTE
*Module/File Name: IGZTSTXF
*******************************************************************************
* Example for callable service CEESTXF *
* COBSTXF - Query current collate category and *
* build input string as function of *
* locale name. *
* Translate string as function of *
* locale. *
* Valid only for COBOL for MVS & VM Release 2 *
* or later. *
*******************************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. COBSTXF.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 MBS.
    02 MBS-Length PIC S9(4) BINARY.
    02 MBS-String PIC X(10).
  01 TXF.
    02 TXF-Length PIC S9(4) BINARY.
    02 TXF-String PIC X(256).
  01 Locale-Name.
    02 LN-Length PIC S9(4) BINARY.
    02 LN-String PIC X(256).
* Use Locale category constants
COPY CEEIGZLC.
*
  01 MBS-Size PIC S9(9) BINARY VALUE 0.
  01 TXF-Size PIC S9(9) BINARY VALUE 0.
  01 FC.
    02 Condition-Token-Value.
      COPY CEEIGZCT.
      03 Case-1-Condition-ID.
        04 Severity PIC S9(4) BINARY.
        04 Msg-No PIC S9(4) BINARY.
      03 Case-2-Condition-ID
        REDEFINES Case-1-Condition-ID.
        04 Class-Code PIC S9(4) BINARY.
        04 Cause-Code PIC S9(4) BINARY.
      03 Case-Sev-Ctl PIC X.
      03 Facility-ID PIC XXX.
      02 I-S-Info PIC S9(9) BINARY.
*
PROCEDURE DIVISION.
*******************************************************************************
* Call CEEQRYL to retrieve locale name *
*******************************************************************************
CALL "CEEQRYL" USING LC-COLLATE,
        Locale-Name, FC.
*******************************************************************************
* Check feedback code and set input string *
*******************************************************************************
IF Severity = 0
```
Locale callable services

IF LN-String (1:LN-Length) = "Sv-SE.IBM-1047"
    MOVE 10 TO MBS-Length
    MOVE 10 TO MBS-Size
    MOVE "7,123,456." TO MBS-String (1:MBS-Length)
ELSE
    MOVE 7 TO MBS-Length
    MOVE 7 TO MBS-Size
    MOVE "8765432" TO MBS-String (1:MBS-Length)
END-IF
ELSE
    DISPLAY "Call to CEEQRYL failed. " Msg-No
    STOP RUN
END-IF.
MOVE SPACES TO TXF-String.
MOVE 0 to TXF-Length.

*************************************************
*  Call CEESTXF to translate the string
*************************************************
CALL "CEESTXF" USING OMITTED, MBS, MBS-Size,
     TXF, TXF-Size, FC.

*************************************************
*  Check feedback code and return length
*************************************************
IF Severity = 0
    IF TXF-Length > 0
        DISPLAY "Translated string is "
        TXF-String
    ELSE
        DISPLAY "String not translated."
    END-IF
ELSE
    DISPLAY "Call to CEESTXF failed. " Msg-No
END-IF.
STOP RUN.
END PROGRAM COBSTXF.

Calls to CEEQRYL and CEESTXF in PL/I

*PROCESS MACRO;
/*Module/File Name: IBMSTXF */
********************************************************************************
/* Example for callable service CEESTXF */
/* Function: Query current collate category and */
/*   build input string as function of locale name. */
/*   Translate string as function of locale. */
********************************************************************************
PLISTXF: PROC OPTIONS(MAIN);
%INCLUDE CEEIBMAW; /* ENTRY defs, macro defs */
%INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants */
%INCLUDE CEEIBMLC; /* Locale category constants */
/* CEESTXF service call arguments */
DCL MBSTRING  CHAR(10) VARYING;   /* input string   */
DCL MBNUMBER  BIN FIXED(31);      /* input length   */
DCL TXFSTRING CHAR(256) VARYING;  /* output string  */
DCL TXFLENGTH BIN FIXED(31);      /* output length  */
/* CEEQRYL service call arguments */
DCL LOCALE_NAME_COLLATE CHAR(256) VARYING;
DCL 01 FC,                        /* Feedback token */
    03 MsgSev    REAL FIXED BINARY(15,0),
    03 MsgNo     REAL FIXED BINARY(15,0),
    03 Flags,    /* Facility ID */
    05 Case      BIT(2),
    05 Severity  BIT(3),
    05 Control   BIT(3),
    03 FacID     CHAR(3),       /* Facility ID */
    03 ISI      /* Instance-Specific Information */
    REAL FIXED BINARY(31,0);
/* retrieve active locale for collate category */
/* Use LC_COLLATE category const from CEEIBMLC */
CALL CEEQRYL ( LC_COLLATE, LOCALE_NAME_COLLATE, FC);
/* FBCHECK macro used (defined in CEEIBMCT) */
IF FBCHECK( FC, CEE000 ) THEN
  DO; /* successful query, set string for CEESTXF */
    IF LOCALE_NAME_COLLATE = 'Sv.SE.IBM-1047' THEN
      MBSTRING = '7,123,456.';
    ELSE
      MBSTRING = '8765432';
    MBNUMBER = LENGTH(MBSTRING);
  END;
ELSE
  DO;
    DISPLAY ( 'Locale LC_COLLATE '||FC.MsgNo );
    STOP;
  END;
TXFSTRING = ';
CALL CEESTXF ( *, MBSTRING, MBNUMBER,
               TXFSTRING, TXFLENGTH, FC );
IF FBCHECK( FC, CEE000 ) THEN
  DO; /* successful call, use transformed length */
    IF TXFLENGTH >0 THEN
      DO;
        PUT SKIP LIST( 'Transformed string is '||
                       SUBSTR(TXFSTRING,1, TXFLENGTH) );
      END;
    END;
ELSE
  DO;
    IF FBCHECK( FC, CEE3TF ) THEN
      DO;
        DISPLAY ( 'Zero length input string' );
      END;
    END;
  END;
END PLISTXF;
Locale callable services
Chapter 23. General callable services

This topic describes the set of Language Environment callable services that provide general services.

List of the general callable services

The general callable services are a set of callable services that are not directly related to a specific Language Environment function.

Related callable services:

- **CEE3DLY**
  - Suspends processing of an active enclave for a specified number of seconds up to a maximum of 1 hour.

- **CEE3DMP**
  - Generates a dump of the Language Environment runtime environment and member language libraries.

- **CEE3USR**
  - Sets or queries one of two 4-byte fields known as the user area fields.

- **CEEDLYM**
  - Suspends processing of an active enclave for a specified number of milliseconds up to a maximum of 1 hour.

- **CEEGPID**
  - Retrieves Language Environment version and platform ID

- **CEEGTJS**
  - Retrieves the value of an exported JCL symbol.

- **CEE3INF**
  - Returns information about the current enclave to the calling routine.

- **CEERAN0**
  - Generates a sequence of uniform pseudorandom numbers between 0.0 and 1.0.

- **CEEEENV**
  - CEEN is a language neutral callable service to get, set, clear, and unset Language Environment variables.

- **CEETEST**
  - Invokes a debug tool, such as the IBM z/OS Debugger.

See z/OS Language Environment Programming Reference for syntax information for callable services.

CEE3DMP callable service

CEE3DMP generates a dump of Language Environment and the member language libraries. Sections of the dump are selectively included, depending on options specified with the options parameter. Output from CEE3DMP is written to the default ddname CEEDUMP, unless you specify the ddname of another file by using the FNAME option of CEE3DMP. The call to CEE3DMP does not cause your application to terminate. For an example of a dump and a description of the Language Environment dump service, see z/OS Language Environment Debugging Guide.

CEE3DMP can be called by your application when you want:

- A trace of calls so you can see the order in which applications were called
- A dump of storage and control blocks
- The status of files to determine if a file is open or closed and to see the buffer contents of the file

General callable services

© Copyright IBM Corp. 1991, 2017
If your application runs in a non-fork() address space, the CEEDUMP DD statement specifies the name of the dump file. If your application runs in the z/OS UNIX environment, CEEDUMP DD can contain the PATH= keyword, which specifies the fully qualified z/OS UNIX 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 (/), and if this directory is writable, and if the CEEDUMP path name (made up of the cwd path name 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

The name of the dump is now CEEDUMP (or the name specified in the FNAME option of CEE3DMP) suffixed with: date.time.process ID.

**CEE3USR callable service**

CEE3USR sets or queries one of two 4-byte fields known as the user area fields. The user area fields are associated with an enclave and are maintained on an enclave basis. A user area might be used by vendor or application programs to store a pointer to a global data area or to keep a recursion counter.

The Language Environment user area fields should not be confused with the PL/I user area. The PL/I user area is a 4-byte field in the PL/I TCA and can only be accessed through assembler language. The PL/I user area continues to be supported for compatibility.

**CEEGPID callable service**

CEEGPID retrieves the Language Environment version ID and the platform ID. The version ID returned by CEEGPID can be tested to determine if you can use new or extended functions that are available in a particular release of Language Environment. For example, POSIX, ENVAR, and CEE3CIB are functions available in Release 3. 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.

**CEEQTJS callable service**

CEEQTJS retrieves the value of an exported JCL symbol.

**CEERANO callable service**

CEERANO generates a sequence of uniform pseudo-random numbers between 0.0 and 1.0 using the multiplicative congruential method with a user-specified seed. The numbers generated are pseudorandom in that the same numbers are generated if the same seed key is used.

**CEEEENV callable service**

CEEEENV is a language neutral callable service to get, set, clear, and unset environment variables. CEEENV processes environment variables depending upon the input function code as follows:

- Obtain the value for an existing environment variable
- Create a new environment variable with a value
- Clear all environment variables
- Delete an existing environment variable
- Overwrite the value for an existing environment variable.
CEE3DLY callable service

CEE3DLY suspends processing of an active enclave for a specified number of seconds up to a maximum of 1 hour.

CEEDLYM callable service

CEEDLYM suspends processing of an active enclave for a specified number of milliseconds up to a maximum of 1 hour.

CEETEST callable service

CEETEST invokes a debug tool, such as the IBM z/OS Debugger. You can use a debug tool to monitor, trace, and interact with your application while it runs. The invocation is dynamic; the debug tool starts when errors are encountered, so you do not have to run your application under an active debug tool.

The z/OS UNIX dbx Debugging Feature, as well as the IBM z/OS Debugger, can be used to debug z/OS XL C/C++ applications that run with POSIX(ON).

Using basic callable services

If you plan to use a Language Environment callable service, you must code a call to the service in your source code, then recompile your source under the latest Language Environment-conforming version of the language you are writing in. The standard call to a Language Environment service is different in each language, but does not differ across operating systems.

The following examples illustrate how the CEEFMDT callable service is called in C, C++, PL/I, and COBOL. CEEFMDT sets the default date and time formats for a specified country. In the examples, country is a 2-character fixed-length string representing a Language Environment-defined country code. Picture string (pic_str or PICSTR) is a character string, containing the default date and time for the country, that is returned by CEEFMDT. A feedback code (fc) returned from the service is checked to determine if the service completed correctly.
/*Module/File Name:  EDCSTRT */
/***************************************************************************/
/*                                                              */
/*  Function:  CEEFMDT - Obtain default date and time format */
/*                                                              */
/***************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

int main(void) {
    _FEEDBACK fc;
    _CHAR2 country;
    _CHAR80 date_pic;
    
    /* get the default date and time format for Canada */
    memcpy(country,"CA",2);
    CEEFMDT(country,date_pic,&fc);
    if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
        printf("CEEFMDT failed with message number %d\n",fc.tok_msgno);
        exit(2999);
    }
    /* print out the default date and time format */
    printf("%.80s\n",date_pic);
}

Figure 91: z/OS XL C/C++ routine with a call to CEEFMDT

The following example shows how the CEEFMDT callable service is called in COBOL.
IDENTIFICATION DIVISION.
PROGRAM-ID. CBLFMDT.

DATA DIVISION.
WORKING-STORAGE SECTION.
  01 COUNTRY PIC X(2).
  01 PICSTR PIC X(80).
  01 FC.
    02 Condition-Token-Value.
    COPY CEEIGZCT.
      03 Case-1-Condition-ID.
      04 Severity PIC S9(4) BINARY.
      04 Msg-No PIC S9(4) BINARY.
      03 Case-2-Condition-ID.
      REDEFINES Case-1-Condition-ID.
      04 Class-Code PIC S9(4) BINARY.
      04 Cause-Code PIC S9(4) BINARY.
      03 Case-Sev-Ctl PIC X.
      03 Facility-ID PIC XXX.
      02 I-S-Info PIC S9(9) BINARY.

PROCEDURE DIVISION.
  PARA-CBLFMDT.
  **************************************************
  ** Specify country code for the US **
  **************************************************
  MOVE "US" TO COUNTRY.
  **************************************************
  ** Call CEEFMDT to return the default date and **
  ** time format for the US **
  **************************************************
  CALL "CEEFMDT" USING COUNTRY, PICSTR, FC.
  **************************************************
  ** If CEEFMDT runs successfully, display result.**
  **************************************************
  IF CEE000 of FC THEN
    DISPLAY "The default date and time "
    "format for the US is: " PICSTR
  ELSE
    DISPLAY "CEEFMDT failed with msg "
    Msg-No of FC UPON CONSOLE
  STOP RUN
  END-IF.
  GOBACK.

Figure 92: COBOL program with a call to CEEFMDT
The following example shows how the CEEFMDT callable service is called in PL/I.

```pli
*PROCESS MACRO;
/*Module/File Name: IBMSTRT
**************************************************************************
/** procedure: CEEFMDT - obtain default  *
/**    date & time format  **/
**************************************************************************
**************************************************************************
PLIFMDT: PROC OPTIONS(MAIN);

%INCLUDE  CEEIBMAW;
%INCLUDE  CEEIBMCT;

DCL COUNTRY CHARACTER ( 2 );
DCL PICSTR  CHAR(80);
DCL 01 FC,                      /* Feedback token */
  03 MsgSev    REAL FIXED BINARY(15,0),
  03 MsgNo     REAL FIXED BINARY(15,0),
  03 Flags,
    05 Case      BIT(2),
    05 Severity  BIT(3),
    05 Control   BIT(3),
  03 FacID     CHAR(3),    /* Facility ID */
    03 ISI                   /* Instance-Specific Information */
    REAL FIXED BINARY(31,0);

COUNTRY = 'US'; /* Specify country code for     
/* the United States            */
/* Call CEEFMDT to get default date format      
/*    for the US                                */
CALL CEEFMDT ( COUNTRY , PICSTR , FC );

/* Print default date format for the US          */
IF FBCHECK( FC, CEE000) THEN DO;
  PUT SKIP LIST( 'The default date and time ' || PICSTR );
END;
ELSE DO;
  DISPLAY( 'CEEFMDT failed with msg ' || FC.MsgNo );
  STOP;
END;

END PLIFMDT;
```

Figure 93: PL/I routine with a call to CEEFMDT

See z/OS Language Environment Programming Reference for detailed instructions on how to call Language Environment services and for more information about the CEEFMDT callable service.
Chapter 24. Math services

This topic introduces Language Environment math services and describes the call interface to the math services.

What Language Environment math services does

Language Environment math services provide standard math computations and can be called from Language Environment-conforming languages or from Language Environment-conforming assembler routines.

You can invoke Language Environment math services by using the call interface or by using the C, COBOL, Fortran, or PL/I built-in math functions specific to the HLL used in your application. For example, your COBOL program can continue to use the built-in SIN function without having to be recoded to use the CEESxSIN call interface.

Language Environment provides several bit manipulation routines to support existing Fortran functions. The scalar versions of Language Environment bit manipulation routines can be called from programs written in any language. For more information on using bit manipulation routines, see z/OS Language Environment Programming Reference.

Related services

Math services:

CEESxABS
  Absolute value

CEESxACS
  Arccosine

CEESxASN
  Arcsine

CEESxATH
  Hyperbolic arctangent

CEESxATN
  Arctangent

CEESxAT2
  Arctangent of two arguments

CEESxCJG
  Conjugate complex

CEESxCOS
  Cosine

CEESxCSH
  Hyperbolic cosine

CEESxCTN
  Cotangent

CEESxDIM
  Positive difference

CEESxDVD
  Division

CEESxERC
  Error function complement
Math services

CEESxERF
   Error function

CEESxEXP
   Exponential (base e)

CEESxGMA
   Gamma function

CEESxIMG
   Imaginary part of a complex

CEESxINT
   Truncation

CEESxLGM
   Log gamma function

CEESxLG1
   Logarithm base 10

CEESxLG2
   Logarithm base 2

CEESxLOG
   Logarithm base e

CEESxMLT
   Floating-point complex multiplication

CEESxMOD
   Modular arithmetic

CEESxNIN
   Nearest integer

CEESxNWN
   Nearest whole number

CEESxSGN
   Transfer of sign

CEESxSIN
   Sine

CEESxSNH
   Hyperbolic sine

CEESxSQT
   Square root

CEESxTAN
   Tangent

CEESxTNH
   Hyperbolic tangent

CEESxXPx
   Exponential (**)

Bit manipulation routines:

CEESICLR
   Bit clear

CEESISET
   Bit set

CEESISHF
   Bit shift

CEESITST
   Bit test
See z/OS Language Environment Programming Reference for syntax and examples of the math services and bit manipulation routines.

### Call interface to math services

The syntax for math services has two forms, depending on how many input parameters the routine requires. The first four letters of the math services are always CEES. The fifth character is $x$, which you replace according to the parameter types listed in “Parameter types: parm1 type and parm2 type” on page 345. The last three letters indicate the math function performed. In these examples, the function performed is the absolute value (ABS).

For one parameter:

\[
\text{CEES}x\text{ABS}(\text{parm1}, \text{fc}, \text{result})
\]

For two parameters:

\[
\text{CEES}x\text{DIM}(\text{parm1}, \text{parm2}, \text{fc}, \text{result})
\]

### Parameter types: parm1 type and parm2 type

The first parameter (parm1) is mandatory. The second parameter (parm2) is used only when you use a math service with two parameters. The $x$ in the fifth space of CEES$x$ must be replaced by a parameter type for input and output. Substitute I, S, D, Q, T, E, or R for $x$:

- **I**: 32-bit binary integer
- **S**: 32-bit single floating-point number
- **D**: 64-bit double floating-point number
- **Q**: 128-bit extended floating-point number
- **T**: 32-bit single floating-complex number (comprised of a 32-bit real part and a 32-bit imaginary part)
- **E**: 64-bit double floating-complex number (comprised of a 64-bit real part and a 64-bit imaginary part)
- **R**: 128-bit extended floating-complex number (comprised of a 128-bit real part and a 128-bit imaginary part)

Language Environment math services expect normalized input. Unless otherwise noted, the result has the same parameter type as the input argument. (For functions of complex variables, the image of the input is generally a nonrectangular shape. For this reason, the output range is not provided.) In the routines described in this topic, the output range for complex-valued functions can be determined from the input range.

C, C++, COBOL, Fortran, and PL/I offer built-in math functions that you can also use under Language Environment.

Simulation of extended-precision floating-point arithmetic is not supported in PL/I routines under CICS.
Examples of calling math services

The following examples illustrate calls to the CEESSLOG math service to calculate the logarithm base e of an argument.

Calling CEESSLOG in C and C++

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

int main (void) {
    float int1, intr;
    _FEEDBACK fc;
    int1 = 39;
    CEESSLOG(&int1,&fc,&intr);
    if ( _FBCHECK ( fc , CEE000 ) != 0 )
    {
        printf("CEESSLOG failed with message number %d\n", fc.tok_msgno);
        exit(2999);
    }
    printf("Log base e of %f is %f\n",int1,intr);
}
```

Figure 94: C/C++ call to CEESSLOG — Logarithm base e
CALLING CEESSLOG IN COBOL

CBL LIB,QUOTE

*************************************************
*Module/File Name: IGZTMATH
*************************************************
** Demonstrates the CEESSLOG math service in COBOL. **
** **************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. MTHSLOG.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 ARG1RS COMP-1.
01 RESLTRS COMP-1.
01 FC.
  02 Condition-Token-Value.
    COPY CEEIGZCT.
    03 Case-1-Condition-ID.
      04 Severity PIC S9(4) BINARY.
      04 Msg-No PIC S9(4) BINARY.
    03 Case-2-Condition-ID.
      REDEFINES Case-1-Condition-ID.
      04 Class-Code PIC S9(4) BINARY.
      04 Cause-Code PIC S9(4) BINARY.
    03 Case-Sev-Ctl PIC X.
    03 Facility-ID PIC XXX.
  02 I-S-Info PIC S9(9) BINARY.
PROCEDURE DIVISION.
PARA-MTHSLOG.
  MOVE 5.65 TO ARG1RS.
  CALL "CEESSLOG" USING ARG1RS, FC, RESLTRS.
  ***************************************************
  ** If CEESSLOG runs successfully, display result. **
  ***************************************************
  IF CEE000 of FC THEN
    DISPLAY "SLOG OF " ARG1RS " = " RESLTRS
  ELSE
    DISPLAY "CEESSLOG failed with msg "
    Msg-No of FC UPON CONSOLE
    STOP RUN
  END-IF.
GOBACK.

Figure 95: Call to CEESSLOG — Logarithm base e in COBOL
Calling CEESSLOG in PL/I

``` PL/I
*PROCESS MACRO;
/*Module/File Name: IBMMATH */
/***********************************************************/
/*                                                        */
/* Demonstrates the CEESSLOG math service in PL/I.        */
/*                                                        */
/***********************************************************/
MTHSLOG: PROC OPTIONS(MAIN);
    %INCLUDE CEEIBMMAW;
    %INCLUDE CEEIBMCT;
    DCL 01 FC, /* Feedback token */
        03 MsgSev    REAL FIXED BINARY(15,0),
        03 MsgNo     REAL FIXED BINARY(15,0),
        03 Flags,   /* Facility ID */
        05 Case      BIT(2),
        05 Severity  BIT(3),
        05 Control   BIT(3),
        03 FacID     CHAR(3), /* Facility ID */
        03 ISI       /* Instance-Specific Information */
            REAL FIXED BINARY(31,0);
    DCL ARG1   REAL FLOAT DECIMAL(6) INIT(5.65);
    DCL RESULT REAL FLOAT DECIMAL(6);
    CALL CEESSLOG (ARG1, FC, RESULT);
    IF FBCHECK( FC, CEE000) THEN
        PUT SKIP LIST( 'SLOG OF ' || ARG1 || ' is ' || RESULT );
    ELSE
        PUT SKIP LIST( 'CEESLOG failed with msg '|| FC.MsgNo );
    END MTHSLOG;
```

Figure 96: Call to CEESSLOG — Logarithm base e in PL/I
Part 4. Using interfaces to other products

Language Environment can be used with applications that run under CICS, Db2, and IMS.

For a list of products that are compatible with Language Environment, see z/OS Language Environment Runtime Application Migration Guide or z/OS Language Environment Concepts Guide.
Chapter 25. Running applications under CICS

Language Environment provides support that, when used in conjunction with facilities provided by the Customer Information Control System (CICS) product, permits you to write applications in high-level languages and run them in a CICS environment. Some of the Language Environment-conforming HLLs might require a later version of CICS. Check the required software list for each language you plan to use.

You can code an application that runs in a CICS environment in Language Environment-conforming assembler, C, C++, COBOL, or PL/I. There is no support for any Fortran routines under CICS. Also, Language Environment-conforming assembler main routines are supported back to z/OS V1R4 when running with CICS transaction Server for z/OS Version 3.1 or later. There is no such restriction on the use of assembler subroutines.

This topic describes special features and considerations that apply to Language Environment-conforming applications running in a CICS environment.

Applications running with POSIX(ON) are only supported in a CICS OTE environment. If you try running an application with POSIX(ON) under non-OTE CICS, it will be overridden with POSIX(OFF) and execution continues. For more information about XPLINK under CICS, go to CICS Transaction Server for z/OS (www.ibm.com/support/knowledgecenter/SSGMGV/welcome.html).

Applications compiled with the XPLINK compiler option are not supported under CICS on releases earlier than CICS TS 3.1. The XPLINK runtime option is ignored under CICS.

CICS does not support PL/I MTF applications.

Terminology used in the Language Environment program model

Before discussing how to develop and run Language Environment-conforming applications in a CICS environment, it is important to map familiar CICS terminology to the terminology used in the Language Environment program model described in Chapter 13, “Program management model,” on page 139.

CICS region

A CICS region is a fixed-size subdivision of main storage allocated to a job step or system task. For example, a CICS region is established during CICS initialization (start-up job). Initialization of a region creates a common environment for all CICS transactions running in that environment. There are no unique Language Environment services that can be applied at a region level.

CICS transaction

A CICS transaction is initiated by a single request, usually from a terminal. A CICS transaction is equivalent to a Language Environment process. A Language Environment process consists of one or more enclaves that carry out the needed processing when they are run. When a CICS transaction is initiated, the first Language Environment thread is triggered within the first enclave in the Language Environment process.

For example, the insertion of a bank card into an ATM machine might trigger a Language Environment process (CICS transaction) consisting of one or more enclaves (CICS run units) to read the information on the card. After an ATM machine reads a bank card, the validation of the information on the card might be performed by one enclave, processing the user’s personal ID number might be performed by another enclave, processing a user request by another, and dispensing the cash by a final enclave.

CICS run unit

A CICS run unit consists of a bound set of one or more load modules that can be loaded by the CICS program loader. Run units are equivalent to Language Environment enclaves. Any link-edited load module is an enclave in CICS; each enclave has its own entry in the CICS PPT. Under CICS, it is possible for a
single enclave to have multiple load modules link-edited with separate entries in the PPT. Each enclave has its own heap storage and other Language Environment resources associated with it.

An enclave is invoked when a Language Environment process (CICS transaction) is triggered or when it is passed control from another enclave using the EXEC CICS LINK or EXEC CICS XCTL commands. For details on using EXEC CICS LINK or EXEC CICS XCTL commands, see "Creating child enclaves with EXEC CICS LINK or EXEC CICS XCTL" on page 470.

Running Language Environment applications under CICS

The following steps describe basic application execution under CICS:

1. An event, generally the receipt of an input message containing a transaction ID code, triggers a Language Environment process (CICS transaction).
2. CICS looks up the transaction ID code in the program control table (PCT) and gets the name of the enclave (or the first enclave) to execute the process.
3. CICS defines the process (transaction) as a work item that is dispatched by the CICS task dispatcher.
4. After the process is defined, CICS looks up the identity of the enclave that is required to perform the task in the processing program table (PPT). The PPT contains information about the enclave such as its language, whether it is in storage, and if in storage, its use count and entry point address.
5. CICS calls the Language Environment-CICS runtime level interface to initialize the process-related portions of the runtime environment.
6. If the enclave does not perform all the processing associated with the process, the enclave might pass control to another enclave through a language call or through the EXEC CICS LINK or EXEC CICS XCTL commands.
7. When the process is complete, CICS calls the Language Environment-CICS runtime level interface to terminate the process-related portions of the runtime environment.

Developing an application under CICS

Certain coding restrictions apply when you develop an application to run under CICS. Examples are:

- Input/output restrictions — CICS provides its own I/O facilities using various EXEC CICS commands.
- Multitasking — CICS has its own multitasking capability.

After you code your application, you must run it through a CICS translator. The translator accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source. The CICS translator runs in a separate job step. The job step sequence for preparing and running an application under CICS is:

1. Code
2. Translate
3. Compile
4. Prelink
5. Link-edit
6. Run

C coding restrictions are described in z/OS XL C/C++ Programming Guide. COBOL restrictions are discussed in the applicable version of the COBOL programming guide in the COBOL library at Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733). Examples of PL/I coding restrictions under CICS are discussed in “PL/I coding considerations under CICS” on page 353.

For more information about developing an application under CICS, go to CICS Transaction Server for z/OS (www.ibm.com/support/knowledgecenter/SSGMGV/welcome.html).
PL/I coding considerations under CICS

CICS imposes some coding restrictions on PL/I routines:

- MTF routines. CICS does not support PL/I MTF applications.
- Built-in subroutines. There are some restrictions on the use of the built-in subroutines of PL/I.
  - You cannot use the PLISRRTx interfaces, PLICKPT or FLICANC.
  - You can use PLIRETC and PLIRETV to communicate between user-written routines that are link-edited together, but not to communicate with CICS. See “Managing return codes in Language Environment” on page 132 for details.
- Debugging facilities. Support has been added to allow the CICS transaction to be debugged using the IBM z/OS Debugger. To prepare your program to use the debug tool, you must compile with the TEST option. For more information on debugging under Language Environment, see z/OS Language Environment Debugging Guide. For more information on debugging CICS transactions using the debug tool, go to the IBM z/OS Debugger (developer.ibm.com/mainframe/products/ibm-zos-debugger).
- Language I/O facilities. You can use only a subset of PL/I I/O facilities under CICS.
  - OPEN/CLOSE can be used, but only for the SYSPRINT file.
  - You can use stream output only to the SYSPRINT file. For performance reasons, you should use stream output under CICS only when debugging your applications.

Those PL/I I/O facilities that you cannot use under CICS are:

- Record I/O statements
- Stream input
- DISPLAY statement
- DELAY statement
- STOP statement
- WAIT statement
- PL/I I/O-related conditions such as RECORD, TRANSMIT, ENDFILE, and KEY are not raised under CICS, because I/O is not performed using PL/I files (except SYSPRINT) and I/O statements. CICS file-handling facilities are used instead. If CICS detects an I/O condition during the processing of your commands, CICS deals with the condition in the way defined in the CICS information.

Assembler considerations

When running with a release of CICS TS earlier than CICS TS 3.1, Language Environment-conforming assembler main routines are not supported under CICS.

Link-edit considerations under CICS

You can link-edit Language Environment-conforming applications that are to be executed under CICS as if they were MVS batch applications. If your C, C++, COBOL, or PL/I application uses EXEC CICS commands, however, you must also link-edit the EXEC CICS interface stub, DFHELII, with your application. To be link-edited with your application, DFHELII must be available in the link-edit SYSLIB concatenation; DFHELII is in the SDFHLOAD library. For more information, go to CICS Transaction Server for z/OS (www.ibm.com/support/knowledgecenter/SSGMGV/welcome.html).

C and C++ considerations

C and C++ applications must be link-edited AMODE(31), RMODE(ANY), as shown in “C/C++ AMODE/ RMODE considerations” on page 12.

COBOL considerations

DFHELII is compatible with the DFHECI stub provided for COBOL programs. Although DFHECI is still supported under Language Environment, DFHELII offers some advantages. Whereas the old COBOL stub
Running applications under CICS

had to be link-edited at the top of your application, DFHELII can be linked anywhere in the application. You also have the capability of linking ILC applications with a single stub rather than with multiple stubs.

**PL/I considerations**

CEESTART is the only entry point for Enterprise PL/I for z/OS and PL/I for MVS & VM applications. To relink OS PL/I object modules with z/OS Language Environment, you must use the following linkage-editor statements:

```
INCLUDE SYSLIB(CEESTART)
INCLUDE SYSLIB(CEESG010)
INCLUDE SYSLIB(DFHELII)
REPLACE PLISTART
CHANGE PLIMAIN(CEEMAIN)
INCLUDE objlib(objmod)
ORDER CEESTART
ENTRY CEESTART
NAME loadmod(R)
```

where:

- **objlib**
  - Represents the PDS that contains the object code

- **objmod**
  - Represents the name of the object module

- **loadmod**
  - Represents the name of the resultant load module

The INCLUDE statement for the object module must occur immediately after the CHANGE statement. The object module of the main procedure must be included before the object modules of subroutines, if any; this was not required for OS PL/I applications.

**CICS processing program table (PPT) considerations**

All of the routines that you dynamically call or fetch in your application must be defined in the CICS PPT. Previously, you had the following choices for the LANG option of the PPT entry: ASM, C, COBOL, and PLI.

Now, however, you can specify 'Language Environment’ (Language Environment) as the language of any Language Environment-conforming routine. This can save you time when you replace a routine with one written in a different language, because you do not need to redefine the routine in the PPT. C++ programs and all Enterprise PL/I for z/OS CICS programs must specify LANG (Language Environment) for the PPT entry.

**Specifying runtime options under CICS**

Under CICS, you cannot pass runtime options as parameters when the application is invoked. However, you can specify runtime options for your application using one of the following methods:

- As system-level default options in a CEEPRMxx parmlib member. For more information, see *z/OS Language Environment Customization*.
- As CICS region-level default options established in CEEROPT. For more information about CEEROPT, see *z/OS Language Environment Customization*.
- In C++ programs, as static ILC to C modules that have a `#pragma runopts` statement
- As application defaults established in CEEUOPT (see “Creating application-specific runtime option defaults with CEEXOPT” on page 105 for details).
- In the user exit. For more information, (see “CEEBXITA assembler user exit interface” on page 378.
- In C applications, as options specified using `#pragma runopts` (see the description in “Methods available for specifying runtime options” on page 101 for information about `#pragma runopts`).
- In PL/I applications, as options specified using the PLIXOPT string (see the description in “Methods available for specifying runtime options” on page 101 for information about the PLIXOPT string).
Some runtime options have different defaults and exhibit slightly different behavior while executing under CICS. The options are listed in Table 57 on page 355.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABPERC</td>
<td>ABPERC is ignored.</td>
</tr>
<tr>
<td>ABTERMENC</td>
<td>ABTERMENC(ABEND) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>AIXBLD</td>
<td>AIXBLD is ignored.</td>
</tr>
<tr>
<td>ALL31</td>
<td>ALL31(ON) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>ANYHEAP</td>
<td>ANYHEAP(4K,4080,ANY,FREE) is the IBM-supplied default under CICS. Both the initial size and the increment size are rounded to the nearest multiple of 8 bytes. In addition, if ANYHEAP(0) is specified, the initial HEAP is obtained on the first use and will be based on the increment size. The maximum initial and increment size for ANYHEAP is 1 gigabyte (1024M).</td>
</tr>
<tr>
<td>ARGPARSE</td>
<td>ARGPARSE is ignored.</td>
</tr>
<tr>
<td>AUTOTASK</td>
<td>AUTOTASK is ignored.</td>
</tr>
<tr>
<td>BELOWHEAP</td>
<td>BELOWHEAP(4K,4080,FREE) is the IBM-supplied default under CICS. Both the initial size and the increment size are rounded to the nearest multiple of 8 bytes. In addition, if BELOWHEAP(0) is specified, then the initial HEAP is obtained on the first use and will be based on the increment size.</td>
</tr>
<tr>
<td>CBLOPTS</td>
<td>CBLOPTS is ignored.</td>
</tr>
<tr>
<td>CBLPSHPOP</td>
<td>ON is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>CBLQDA</td>
<td>CBLQDA is ignored.</td>
</tr>
<tr>
<td>CHECK</td>
<td>ON is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>COUNTRY</td>
<td>The value specified is user-defined.</td>
</tr>
<tr>
<td>DEBUG</td>
<td>DEBUG is ignored.</td>
</tr>
<tr>
<td>DEPTHCONDLMT</td>
<td>For C, COBOL, FORTRAN, and applications with multiple languages, the recommended value is 10. For PL/I, the recommended value is 0.</td>
</tr>
<tr>
<td>ENV</td>
<td>ENV is ignored.</td>
</tr>
<tr>
<td>ENVAR</td>
<td>ENVAR sets the initial value for environment variables. With ENVAR, you can pass switches or tagged information into the application using standard z/OS UNIX functions getenv(), setenv(), and clearenv().</td>
</tr>
<tr>
<td>ERRCOUNT</td>
<td>ERRCOUNT(0) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>ERRUNIT</td>
<td>ERRUNIT is ignored.</td>
</tr>
<tr>
<td>EXECOPS</td>
<td>EXECOPS is ignored.</td>
</tr>
<tr>
<td>FILEHIST</td>
<td>FILEHIST is ignored.</td>
</tr>
<tr>
<td>FILETAG</td>
<td>FILETAG is ignored.</td>
</tr>
<tr>
<td>FLOW</td>
<td>NOFLOW is the IBM-supplied default under CICS.</td>
</tr>
</tbody>
</table>
### Table 57: Runtime option behavior under CICS (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HEAP</strong></td>
<td>HEAP(4K,4080,ANY,KEEP,4K,4080) is the IBM-supplied default under CICS. Both the initial size and the increment size are rounded to the next multiple of 8 bytes. In addition, if HEAP(0) is specified, then the initial HEAP is obtained on the first use and will be based on the increment size. The maximum initial and increment size for HEAP is 1 gigabyte (1024M).</td>
</tr>
<tr>
<td><strong>HEAPCHK</strong></td>
<td>HEAPCHK(OFF,1,0,0,0,1024,0,1024,0) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td><strong>HEAPPOLS</strong></td>
<td>HEAPPOLS(OFF,8,10,32,10,128,10,256,10,1024,10,2048,10,0,10,0,10,0,10,1,0,0,10,2048,10,0,10,0,10,0,10) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td><strong>HEAPZONES</strong></td>
<td>HEAPZONES(0,ABEND,0,ABEND) is the IBM-supplied default under CICS</td>
</tr>
<tr>
<td><strong>INQPCOPN</strong></td>
<td>INQPCOPN is ignored.</td>
</tr>
<tr>
<td><strong>INTERRUPT</strong></td>
<td>INTERRUPT is ignored.</td>
</tr>
<tr>
<td><strong>LIBSTACK</strong></td>
<td>LIBSTACK(32,4080,FREE) is the IBM-supplied default under CICS. Both the initial size and the increment size are rounded to the nearest multiple of 8 bytes. The minimum initial size is 32 bytes; the minimum increment size is 4080 bytes. When ALL31 is ON, LIBSTACK is not allocated below the 16M line.</td>
</tr>
<tr>
<td><strong>MSGFILE</strong></td>
<td>MSGFILE is ignored. All messages and output (dumps and reports, for example) are sent to a transient data queue called CESE (for more information, see “Runtime output under CICS” on page 362 ).</td>
</tr>
<tr>
<td><strong>MSGQ</strong></td>
<td>15 is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td><strong>NATLANG</strong></td>
<td>ENU is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td><strong>NONIPTSTACK</strong></td>
<td>NONIPTSTACK is ignored.</td>
</tr>
<tr>
<td><strong>OCSTATUS</strong></td>
<td>OCSTATUS is ignored.</td>
</tr>
<tr>
<td><strong>PAGEFRAMESIZE</strong></td>
<td>PAGEFRAMESIZE is ignored. Note the statement of direction associated with this runtime option.</td>
</tr>
<tr>
<td><strong>PC</strong></td>
<td>PC is ignored.</td>
</tr>
<tr>
<td><strong>PLIST</strong></td>
<td>PLIST is ignored.</td>
</tr>
<tr>
<td><strong>PLITASKCOUNT</strong></td>
<td>PLITASKCOUNT is ignored.</td>
</tr>
<tr>
<td><strong>POSIX</strong></td>
<td>POSIX is ignored.</td>
</tr>
<tr>
<td><strong>PROFILE</strong></td>
<td>PROFILE(OFF) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td><strong>PRTUNIT</strong></td>
<td>PRTUNIT is ignored.</td>
</tr>
<tr>
<td><strong>PUNUNIT</strong></td>
<td>PUNUNIT is ignored.</td>
</tr>
<tr>
<td><strong>RDRUNIT</strong></td>
<td>RDRUNIT is ignored.</td>
</tr>
<tr>
<td><strong>RECPAD</strong></td>
<td>RECPAD is ignored.</td>
</tr>
<tr>
<td><strong>REDIR</strong></td>
<td>REDIR is ignored.</td>
</tr>
<tr>
<td><strong>RTEREUS</strong></td>
<td>RTEREUS is ignored.</td>
</tr>
<tr>
<td><strong>SIMVRD</strong></td>
<td>SIMVRD is ignored.</td>
</tr>
<tr>
<td><strong>STACK</strong></td>
<td>STACK(4K,4080,ANY,KEEP,..) is the IBM-supplied default under CICS.</td>
</tr>
</tbody>
</table>
Table 57: Runtime option behavior under CICS (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE</td>
<td>STORAGE(NONE,NONE,NONE,0K) is the IBM-supplied default under CICS. Your</td>
</tr>
<tr>
<td></td>
<td>application could require a different setting. For example, to initialize</td>
</tr>
<tr>
<td></td>
<td>memory to zeros, use STORAGE(00,00,00).</td>
</tr>
<tr>
<td>TERMTHDACT</td>
<td>All TERMTHDACT output (including that from dumps) is written to a transient</td>
</tr>
<tr>
<td></td>
<td>data queue named CESE.</td>
</tr>
<tr>
<td>TEST</td>
<td>NOTEST(ALL,*,PROMPT,INSPREF) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>THREADHEAP</td>
<td>THREADHEAP is ignored.</td>
</tr>
<tr>
<td>THREADSTACK</td>
<td>THREADSTACK is ignored.</td>
</tr>
<tr>
<td>TRACE</td>
<td>TRACE(OFF,4K,DUMP,LE=0) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>TRAP</td>
<td>TRAP(ON,SPIE) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>UPSI</td>
<td>UPSI(00000000) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>USERHDLR</td>
<td>The specified value is user-defined.</td>
</tr>
<tr>
<td>VCTRSAVE</td>
<td>VCTRSAVE(OFF) is the IBM-supplied default under CICS.</td>
</tr>
<tr>
<td>XPLINK</td>
<td>XPLINK is ignored.</td>
</tr>
<tr>
<td>XUFLOW</td>
<td>XUFLOW(AUTO) is the IBM-supplied default under CICS.</td>
</tr>
</tbody>
</table>

Accessing DLI databases from CICS

Various user interfaces to DLI databases on IMS are available under CICS. See Appendix B, “EXEC DLI and CALL IMS Interfaces,” on page 499 for details.

Using callable services under CICS

All Language Environment callable services are available to applications executing as CICS transactions. However, the CEEMOUT (dispatch a message) and CEE3DMP (generate dump) services differ, in that the messages and dumps are sent to a transient data queue called CESE rather than to the ddname specified in the MSGFILE runtime option. See z/OS Language Environment Programming Reference for descriptions of these services.

See z/OS Language Environment Writing Interlanguage Communication Applications for ILC examples that make a call to CEEMOUT.

OS/VS COBOL compatibility considerations under CICS

OS/VS COBOL programs, which had runtime support in CICS® Transaction Server for z/OS®, Version 2, cannot run under CICS TS for z/OS, Version 3 or later.

Using math services in PL/I under CICS

Simulation of extended-precision floating-point arithmetic is not supported in PL/I routines under CICS.

PL/I saves and restores floating-point registers where necessary. PLIDUMP can print these registers (for more information about PLIDUMP, see z/OS Language Environment Debugging Guide).

Floating-point overflow and underflow can be handled in OVERFLOW and UNDERFLOW ON-units. The program mask is set appropriately for the levels of CICS and PL/I used.
Coding program termination in PL/I under CICS

You can terminate a PL/I routine running under CICS by using PL/I constructs or CICS statements such as EXEC CICS RETURN, EXEC CICS SEND PAGE RELEASE, EXEC CICS XCTL, or EXEC CICS ABEND. When the routine terminates, the following occurs:

1. If you requested a storage report using the RPTSTG runtime option, the report is written to the CESE transient data queue (described in “Runtime output under CICS” on page 362).
2. If CESE is still open, it is closed.
3. All storage acquired by PL/I is freed before control returns to CICS, except for the stack.

Storage management under CICS

Applications can allocate and free storage explicitly through language facilities, CICS facilities or the Language Environment storage management callable services. For more information about the EXEC CICS GETMAIN and FREEMAIN commands, go to CICS Transaction Server for z/OS (www.ibm.com/support/knowledgecenter/SSGMGV/welcome.html).

If you do not explicitly free storage that was allocated through language facilities or Language Environment callable services, the storage is freed at enclave termination.

CICS short-on-storage condition

The CICS short-on-storage condition might be raised under Language Environment if functions in your application attempt to acquire storage by using language facilities and not enough storage is available to satisfy the request. CICS places the transaction on a queue until the storage request can be satisfied. If CICS cannot get enough storage in a reasonable amount of time to satisfy the request, then the transaction that issued the storage request is terminated by CICS with abend code AKCP.

CICS storage protect facility

The CICS Storage Protect Facility allows you to isolate user applications from CICS storage. For information about the CICS Storage Protect Facility, go to CICS Transaction Server for z/OS (www.ibm.com/support/knowledgecenter/SSGMGV/welcome.html).

All storage that Language Environment acquires on behalf of the user, such as working storage, heap, and stack, it acquires in the default key specified on the transaction. All storage that Language Environment acquires for its own use, such as control blocks, it acquires using USERDATAKEY storage. Applications running with Language Environment can obtain CICSDATAKEY storage by using the EXEC CICS interface.

PL/I storage considerations under CICS

Special storage considerations for running PL/I applications under CICS are described in the following topics

Initializing static external data

You must initialize static external data under CICS because CICS cannot handle common CSECTs.

PL/I object program size

The load module resulting from a PL/I application must not occupy more than 524,152 bytes of main storage. An exception is that an RMODE=ANY program can occupy 16 megabytes, although this is not recommended.

Using CICS storage constructs instead of PL/I language statements

In the case when a PL/I routine (routine A, for example) issues an EXEC CICS LINK to another PL/I routine (routine B, for example), you might want to use EXEC CICS GETMAIN and FREEMAIN commands to get and free storage. This is because the scope of EXEC CICS GETMAIN is the scope of the entire task, not just a single routine. Either routine A or routine B can explicitly free the storage. Alternatively, you can choose
to not explicitly free the storage in either routine, but allow the storage to be freed automatically when the
task is terminated. Another advantage to using EXEC CICS GETMAIN is that if routine A terminates, the
storage is still available to routine B.

When you use PL/I language statements to get and free storage, the scope of PL/I storage statements is
the routine, not the task. Although routine B can alter the storage allocated by routine A by using a
pointer, routine B cannot free the storage. In addition, if routine A terminates, the storage is automatically
freed. Routine B can no longer access the storage.

**PL/I storage classes**

When using CICS, you should avoid altering STATIC storage. Doing so violates reentrancy and can yield
unpredictable results. Instead of altering STATIC storage, you should make most or all user variables that
are changed while the routine is running AUTOMATIC. Those user variables with initial values that never
change should be declared STATIC INITIAL.

Although AUTOMATIC storage provides reentrancy and should suffice for most purposes, you can also
allocate and free storage with the ALLOCATE and FREE statements, which you can use to allocate and free
BASED and CONTROLLED variables. References you make to BASED storage are handled with the pointer
set by the ALLOCATE statement. The pointer itself can be AUTOMATIC.

You can use CONTROLLED storage under CICS, because it is consistent with reentrancy.

**Using PUT DATA with BASED storage**

BASED storage is used extensively in CICS transactions. You therefore need to be aware of the following
restriction on PUT DATA.

In PL/I, you cannot code:

```
PUT DATA (P -> VAR);
```

If, however, VAR was declared as BASED (P), the value of the generation of VAR to which P points can be
coded as:

```
PUT DATA (VAR);
```

**Using STORAGE built-in functions**

The STORAGE and CURRENTSTORAGE built-in functions return the length of an item to your PL/I routine.
This is useful in CICS, where functions often require the length of an argument as well as its address. In
particular, you can use these functions to get lengths of PL/I aggregates without having to count or
compute such lengths or specify length fields in the CICS commands.

For more information about the STORAGE and CURRENTSTORAGE built-in functions, see the IBM

---

**Condition handling under CICS**

The Language Environment condition handling services described in Chapter 15, “Introduction to
Language Environment condition handling,” on page 167 and in other topics are supported under CICS,
but additional considerations apply when running an application under CICS; these considerations are
described in the following topics.

Condition handling in nested enclaves created by EXEC CICS LINK or EXEC CICS XCTL is discussed in
“How conditions arising in child enclaves are handled” on page 470.

**PL/I considerations for using the CICS HANDLE ABEND command**

The EXEC CICS HANDLE facility resembles a PL/I ON-unit with this syntax:

```
ON condition GO TO label;
```
You can code the HANDLE command wherever you would code the ON...GO TO...statement. The label to be branched to can be located in any other active block, and the condition can arise in an even later block. HANDLE terminates intervening PL/I blocks by invoking PL/I's out-of-block GO TO facilities.

**Note:** Because PL/I internal procedures are not active at all times, you should not use internal procedures as exit routines in HANDLE commands.

HANDLE is not semantically identical to the ON condition GO TO label; statement. A PL/I ON-unit disappears when the block containing it terminates; a CICS HANDLE disappears permanently when it is explicitly overridden by another one at the same logical LINK level.

A HANDLE command could specify a branch to a label in a block no longer active. Because HANDLE is implemented by forcing a PL/I out-of-block GO TO, this is equivalent to assigning a label constant to a PL/I label variable after the block containing the label constant has terminated, which is invalid. The PL/I out-of-block GO TO mechanism attempts to detect this error and raises the ERROR condition. If PL/I out-of-block GO TO fails to detect such an invalid GO TO, however, the GO TO becomes a wild branch that causes some unpredictable failure. Thus, upon return from a PL/I block that established HANDLE for a particular condition, your program should issue a resetting HANDLE for that condition (provided, of course, that there is still some possibility of the condition arising). A PL/I ON-unit does not have to be reset.

**Effect of the CICS HANDLE ABEND command**

When an application is running under CICS with Language Environment, condition handling differs depending on whether a CICS HANDLE ABEND is active or not active.

When a CICS HANDLE ABEND is active, Language Environment condition handling does not gain control for any abends or program interrupts. Any abends or program interrupts that occur while a CICS HANDLE ABEND is active cause the action defined in the CICS HANDLE ABEND to take place. The user-written condition handlers established by CEEHDLR are ignored.

When a CICS HANDLE ABEND is not active, Language Environment condition handling does gain control for abends and program interrupts if the TRAP(ON) option is specified. Normal Language Environment condition handling is then performed.

**Effect of the CICS HANDLE CONDITION and CICS HANDLE AID**

Language Environment condition handling does not alter the behavior of applications that use CICS HANDLE CONDITION or CICS HANDLE AID. The CICS CONDITION and AID conditions are raised by CICS and are handled only by CICS; Language Environment is not involved in the handling of CICS conditions.

**Restrictions on user-written condition handlers under CICS**

The following EXEC CICS commands cannot be used within a user-written condition handler established using CEEHDLR, or within any routine called by the user-written condition handler:

- EXEC CICS ABEND
- EXEC CICS HANDLE AID
- EXEC CICS HANDLE ABEND
- EXEC CICS HANDLE CONDITION
- EXEC CICS IGNORE CONDITION
- EXEC CICS POP HANDLE
- EXEC CICS PUSH HANDLE

All other EXEC CICS commands are allowed within a user-written condition handler. However, they must be coded using the NOHANDLE option, the RESP option, or the RESP2 option. This prevents additional conditions being raised due to a CICS service failure.
COBOL considerations

A user-written condition handler registered for a routine using the CEEHDLR service cannot be translated using the CICS translator and therefore cannot contain any EXEC CICS commands. This is because the CICS translator inserts (onto the PROCEDURE DIVISION header of the COBOL program) the arguments EXEC Interface Block (EIB) and COMMAREA, which do not match arguments passed by Language Environment.

However, a user-written condition handler can call a subroutine to perform EXEC CICS commands. If arguments need to be passed to this subroutine, they should be preceded by two dummy arguments in the caller. The called subroutine must issue EXEC CICS ADDRESS EIB before executing any other EXEC CICS commands.

CICS transaction abend codes

The same Language Environment reserved abend codes (4000 through 4095) are used for applications running under CICS. In addition, there are special reason codes returned to CICS for severe Language Environment conditions. These severe conditions are CICS-specific. For a detailed explanation of these reason codes, see z/OS Language Environment Runtime Messages.

Using the CBLPSHPOP runtime option under CICS

This topic applies to Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II programs only

The CBLPSHPOP runtime option controls whether the Language Environment environment automatically issues an EXEC CICS PUSH HANDLE command during initialization and an EXEC CICS POP HANDLE command during termination whenever a COBOL subroutine is called.

If your application calls COBOL subroutines under CICS, your application performance is better with CBLPSHPOP(OFF) than with CBLPSHPOP(ON). You can set CBLPSHPOP on a transaction-by-transaction basis by using CEEUOPT.

For more information about CBLPSHPOP, see z/OS Language Environment Programming Reference.

Restrictions on assembler user exits under CICS

The following EXEC CICS commands cannot be used within the assembler user exit or any routines called by the assembler user exit:

- EXEC CICS ABEND
- EXEC CICS HANDLE AID
- EXEC CICS HANDLE ABEND
- EXEC CICS HANDLE CONDITION
- EXEC CICS PUSH HANDLE
- EXEC CICS POP HANDLE
- EXEC CICS IGNORE CONDITION

All other EXEC CICS commands are allowed within the assembler user exit. However, they must be coded using the NOHANDLE option, the RESP option, or the RESP2 option. This prevents additional conditions being raised due to a CICS service failure.


PL/I considerations

You can use PLIRETC to communicate with the Language Environment assembler user exit. For more information about PLIRETC, see “For PL/I” on page 134. For more information about the assembler user exit, see Chapter 28, “Using runtime user exits,” on page 373.
The CICS user exit for PL/I, IBMFXITA, is supported for compatibility by Language Environment. For migration information, see IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

### Ensuring transaction rollback under CICS

Conditions that occur while an application is executing under CICS can potentially contaminate any database currently being used by the application. It is essential that a rollback (the backing out of any updates made by the failing application) be performed before further damage to the database can occur.

There are two ways to ensure that a transaction rollback occurs when an unhandled condition of severity 2 or greater is detected:

- Use the ABTERMENC(ABEND) runtime option, or
- Make sure the assembler user exit requests an abend for unhandled conditions of severity 2 or greater.

See z/OS Language Environment Programming Reference for an explanation of the ABTERMENC runtime option. For more information about using assembler user exits, see Chapter 28, “Using runtime user exits,” on page 373.

### Runtime output under CICS

Language Environment provides the same message handling and dump services for CICS as it does for non-CICS systems. Any exceptions to this support under CICS are noted in the following topics.

### Message handling under CICS

The MSGFILE runtime option is ignored under CICS, because messages for a run unit are directed instead to the CICS transient data queue named CESE.

Messages are prefixed by a terminal ID, a transaction ID, a date, and a timestamp before their transmission. Figure 97 on page 362 illustrates this format.

<table>
<thead>
<tr>
<th>ASA</th>
<th>Terminal ID</th>
<th>Transaction ID</th>
<th>sp</th>
<th>Time Stamp</th>
<th>sp</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>132</td>
</tr>
</tbody>
</table>

*Figure 97: Format of messages sent to CESE*

ASA  

Terminal ID  
A 4-character terminal identifier.

Transaction ID  
A 4-character transaction identifier.

sp  
A space.

Timestamp  
The date and time displayed in the same format as that returned by the CEELOCT service.

Message  
The message identifier and message text.
The entire message record is preceded by an ASCII control character to determine the format of the printing.

Message records are V-format.


**PL/I SYSPRINT**

PL/I SYSPRINT also uses the CESE transient data queue. For information on how to declare SYSPRINT, see the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

**CICS XPLINK SYSPRINT**

CICS XPLINK applications use SYSPRINT as the destination when writing to the COUT, CEEOUT, or STDOUT data streams.

**Dump services under CICS**

Under CICS, the FNAME parameter of the CEE3DMP callable service is ignored. Instead of being written to a ddname specified in FNAME, dumps are instead transmitted to the CICS transient data queue named CESE.

The dump is prefixed with the same information shown in Figure 97 on page 362.

**PL/I considerations**

The PLIDUMP subroutine has two additional options under CICS and some special considerations. For more information about PLIDUMP, see z/OS Language Environment Debugging Guide.

**Support for calls within the same HLL under CICS**

For ILC information while running under CICS, see z/OS Language Environment Writing Interlanguage Communication Applications.

**C**

EXEC CICS LINK, EXEC CICS XCTL, and calls via fetch() and DLL are supported under CICS. The fetched program or DLL must be defined in the CSD and installed in the PPT. For more information, see z/OS XL C/C++ Programming Guide.

**C++**

EXEC CICS LINK, EXEC CICS XCTL, and dynamic calls via DLL are supported under CICS. The DLL must be defined in the CSD and installed in the PPT. For further information, see z/OS XL C/C++ Programming Guide.

**COBOL**

The following topics describe support for calls compiled under different versions of COBOL compilers.

**Language Environment-conforming COBOL**

Static and dynamic calls between Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II programs are supported as follows:

- Called programs can contain any command or facility supported by CICS for COBOL.
- If the called program has been translated by the CICS translator, calling programs must pass the EIB and COMMAREA as the first two parameters on the CALL statement.
Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM and COBOL/370 programs can invoke or be invoked by VS COBOL II programs only through CICS facilities such as EXEC CICS LINK, EXEC CICS XCTL, and EXEC CICS RETURN.

**VS COBOL II**
Static and dynamic calls to or from Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, and VS COBOL II programs are supported with the same considerations that were previously listed.

VS COBOL II programs can communicate with OS/VS COBOL programs only through CICS facilities such as EXEC CICS LINK, EXEC CICS XCTL, and EXEC CICS RETURN.

**OS/VS COBOL**
OS/VS COBOL programs cannot directly call or be called from Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, or VS COBOL II programs. The only programs that can be called from OS/VS COBOL are other OS/VS COBOL programs, and with the following restrictions:

- The call must be a static CALL
- The subprogram cannot contain EXEC CICS statements.

**PL/I**
Static calls are supported from any version of PL/I. Called subroutines can invoke CICS services if the address of the EIB is passed to the subroutine properly. You can do call the subroutines by setting up the address of the EIB yourself and passing it to the subroutine, or by coding the following command in the subroutine before you issue any other CICS commands.

```plaintext
EXEC CICS ADDRESS EIB(DFHEIPTR)
```

PL/I FETCH is supported under CICS in a PL/I transaction that is compiled with Enterprise PL/I for z/OS and PL/I for MVS & VM. CICS does not support PL/I MTF applications.
Chapter 26. Running applications under Db2

Language Environment supports Db2 applications.

An application program requests Db2 services by using SQL statements imbedded in the program. The imbedded SQL is translated by the SQL precompiler into host language statements that typically perform assignments and then call a Db2 language interface module. The same entry point of the module is called by all Language Environment-conforming languages. Db2 processes the request and then returns to the application.

Language Environment support for Db2 applications

You are not required to modify anything in your code to run a Language Environment-conforming application with Db2. Language Environment also supports ILC applications that use Db2 services.

Language Environment supports XPLINK applications that use Db2 services, but does not support stored Db2 procedures compiled with XPLINK.

Condition handling under Db2

Any errors occurring in Db2 are trapped by Db2 and handled properly. If a task terminates, Db2 takes appropriate action depending on the nature of termination.

The Language Environment runtime user exits can be used by the installation to effect abnormal termination, and therefore database rollbacks, of applications.

For information about additional HLL restrictions under Db2, see the Application Programming Guide for your HLL. For more information about using Db2 services, see Database 2 Application Programming and SQL Guide.

PL/I consideration for Db2 applications

PL/I multitasking applications are not supported under Db2. PL/I multitasking applications support Db2 SQL statements from multiple tasks only in non-CICS and non-IMS environments.
Chapter 27. Running applications under IMS

This topic describes Language Environment support for applications running under IMS/ESA Version 3 Release 1 and later.

You do not need to change any of the code in your application in order to run under IMS/ESA, but there are a number of restrictions and recommendations that you should consider. Two of these concerns include ensuring proper condition handling under IMS and running your application in an IMS/ESA environment. These topics, together with an overview of how Language Environment interacts with IMS, are discussed in detail.

For a detailed description of how to write IMS batch and online applications, see the IMS Application Programming Guide appropriate to your version of IMS.

Using the interface between Language Environment and IMS

Language Environment provides a callable service, CEETDLI, that you can use to invoke IMS (Version 4 or later) facilities. In assembler, COBOL, PL/I, C and C++, you can also invoke IMS by using the following interfaces:

- In assembler, the ASMTDLI interface
- In COBOL, the CBLTDLI interface
- In PL/I, the PLITDLI interface
- In C or C++, the CTDLI interface (a ctdli() function call)
- In C or C++ (including XPLINK-compiled functions), the CTDLI interface (a ctdli() function call)

Under Language Environment, each of these interfaces continues to function in its current capacity. CEETDLI performs essentially the same functions, but it offers some advantages, particularly if you plan to run an ILC application in IMS. For example, if you use CEETDLI, you get coordinated condition handling between Language Environment and IMS condition handling facilities. For more information, see “Coordinated condition handling under IMS” on page 369.

CEETDLI supports calls that use an application interface block (AIB) or a program communication block (PCB).

For more information about AIB and a complete description of all available IMS functions and argument parameters you can specify in CEETDLI, see an IMS Application Programming Guide.

Appendix B, “EXEC DLI and CALL IMS Interfaces,” on page 499 lists various DL/I interfaces and the support for them under CICS and IMS. For information about CEETDLI, including its syntax and examples, see z/OS Language Environment Programming Reference.

z/OS XL C/C++ considerations under IMS

To interface with IMS from z/OS XL C/C++, you must do the following:

- Specify the PLIST(OS), ENV(IMS), and NOEXECOPS runtime options of #pragma runopts in your source code. The PLIST(OS) option establishes the correct parameter list format when invoked under IMS. The ENV(IMS) option establishes the correct operating environment. The NOEXECOPS option specifies that runtime options cannot be specified for IMS.
- When you use the PLIST(OS) option in #pragma runopts, argc contains 1 (one) and argv[0] contains NULL.

For more information about using the #pragma runopts preprocessor directive, see Chapter 9, “Using runtime options,” on page 101.
Applications that use the POSIX(ON) runtime option are supported under IMS only if they consist of a single thread. Calls to z/OS UNIX threading functions are restricted under IMS. See z/OS XL C/C++ Programming Guide for a list of restrictions on running z/OS XL C/C++ programs under IMS with z/OS UNIX.

The IMS environment supports 31-bit XPLINK applications. However, applications that make many calls to the Language Environment callable service CEETDLI or the C ctdli() function might not be suitable for XPLINK because of the overhead for these XPLINK to nonXPLINK calls. See “When XPLINK should not be used” on page 32 for more information.

C++ considerations under IMS

To interface with IMS from C++, you must do the following:

- For any C++ program that runs under IMS, you must specify the TARGET(IMS) compiler option.
- For any C++ program that is the initial main() called under IMS, you must specify the PLIST(OS) compiler option.
- For any C++ programs in nested enclaves, you need only specify the TARGET(IMS) compiler option.

PL/I considerations under IMS

With IMS/ESA Version 4, PL/I supports PSBs with LANG=PLI and all others (including LANG=blank), except LANG=Pascal. With IMS/ESA Version 3 Release 1, PL/I supports PSBs with LANG=PLI only.

The SYSTEM(IMS) compiler option must be specified for PL/I applications running under IMS. When SYSTEM(IMS) is specified, the OPTIONS(BYVALUE) attribute is implied for any external PROCEDURE that also has OPTIONS(MAIN). Further, the parameters to such a MAIN procedure must be POINTERS.

With IMS/ESA Version 3 Release 1 and Version 4, the parameters passed to language-IMS CALL interfaces such as PLITDLI are no longer required to be below the 16M line.

If an assembler program is driving a transaction program written in Enterprise PL/I for z/OS or PL/I for MVS & VM, the main procedure of the transaction must be compiled with SYSTEM(MVS) option; the main procedure receives the parameter list passed from the assembler program in MVS style. IMS does not support PL/I MTF applications.

IMS communication with your application

When you run your application under IMS, IMS loads the application and passes it the invocation parameter list. A PSB is automatically scheduled for every application you run under IMS. IMS/ESA Version 4 always constructs the parameter list in the same format, regardless of the setting of the LANG= option in the PSB. The LANG= option has no effect on the format of the parameter list that IMS constructs. Thus, any PSB can be used with any HLL application in Language Environment.

Beginning with IMS/ESA Version 4, the LANG= option in the PSB statement is not required.

Before your application is loaded, it is link-edited with an IMS language interface module, DFSLI000. Any calls that your application makes with CEETDLI for IMS services end up in this module. DFSLI000, in turn, invokes the services and returns IMS replies to your application.

Link-edit considerations under IMS

Unless your application communicates with IMS exclusively via dynamic calls to the CEETDLI callable service, DFSLI000 must be link-edited with your application code. Therefore, under MVS, ensure that DFSLI000 appears in a partitioned data set that is specified in the SYSLIB DD statement in the JCL used to link-edit the application.
You must be using IMS Version 4 or later if you want to use the CEETDLI callable service. Errors occur if you attempt to use the CEETDLI callable service and you are not running IMS Version 4.

Making your IMS application reentrant

For many IMS users, the storage required at run time for any given IMS transaction can be fairly large. Therefore, the most efficient method of coding an application is to make it reentrant. This method can eliminate application loading time, speed up IMS initialization and restart, and provide the additional integrity that results from having your routines in protected storage.

Methods for making your application reentrant differ across HLLs. For more information, see Chapter 11, “Making your application reentrant,” on page 121.

Condition handling under IMS

The IMS environment is sensitive to errors or conditions. A failing IMS transaction or application can potentially contaminate an IMS database. For this reason, IMS must know about the failure of a transaction or application that has been updating a database so that it can perform database rollback (the backing out of any updates made by the failing application).

Coordinated condition handling under IMS

Language Environment and IMS condition handling is coordinated, meaning that if a program interrupt or abend occurs when your application is running in an IMS environment, the Language Environment condition manager can determine whether the problem occurred in your application or in IMS. If the program interrupt or abend occurs in IMS, Language Environment, as well as any invoked HLL-specific condition handler, percolates the condition back to IMS.

If a program interrupt or abend occurs in the application outside of IMS, or if a software condition of severity 2 or greater is raised outside of IMS, the Language Environment condition manager takes normal condition handling actions as described in Chapter 15, “Introduction to Language Environment condition handling,” on page 167. If the condition manager remains in control, however, you must do one of the following:

• Resolve the error so that the application can continue.
• Issue a rollback call to IMS, and then terminate the application.
• Ensure that the application terminates abnormally by using the ABTERMENC(ABEND) runtime option to transform all abnormal terminations into operating system abends in order to cause IMS rollbacks.
• Ensure that the application terminates abnormally by coding and providing a modified runtime assembler user exit (CEEBXITA) that transforms all abnormal terminations into operating system abends in order to cause IMS rollbacks.

The assembler user exit you provide should check the return code and reason code or the CEEAUE_ABTERM bit, and request an abend by setting the CEEAUE_ABND flag to ON, if appropriate. See “CEEBXITA assembler user exit interface” on page 378 for more details about the CEEBXITA user exit.

Diagnosing abends with the IMS dump

If an interrupt or abend occurs in IMS, you can use the IMS dump (which contains the information that is available at the time of the program interrupt or abend) for diagnosis. You can also use the Language Environment dump (CEEDUMP) for diagnosis.

If the interrupt or abend occurs in your application (outside of IMS), the IMS dump shows the state of the system after Language Environment gained control, did some cleanup of the environment, and requested the abend. In this case, you can use only the Language Environment dump (CEEDUMP) for diagnosis.
Part 5. Specialized programming tasks

This section describes advanced or specialized tasks that you can perform in Language Environment.
Chapter 28. Using runtime user exits

Language Environment provides user exits that you can use for functions at your installation. You can use the assembler user exit (CEEBXITA) or the HLL user exit (CEEBINT). This Using nested enclaves provides information about using these runtime user exits.

User exits are invoked under Language Environment to perform enclave initialization functions and both normal and abnormal termination functions. User exits offer you a chance to perform certain functions at a point where you would not otherwise have a chance to do so. In an assembler initialization user exit, for example, you can specify a list of runtime options that establish characteristics of the environment. This is done before the actual execution of any of your application code.

In most cases, you do not need to modify any user exit in order to run your application. Instead, you can accept the IBM-supplied default versions of the exits, or the defaults as defined by your installation. To do so, run your application in the normal manner and the default versions of the exits are invoked. You might also want to read “User exits supported under Language Environment” on page 373 and “When user exits are invoked” on page 375, which provide an overview of the user exits and describe when they are invoked.

If you plan to modify either of the user exits to perform some specific function, you must link the modified exit to your application before running. In addition, “Using the assembler user exit CEEBXITA” on page 374 and “CEEBINT high-level language user exit interface” on page 387 describe the respective user exit interfaces to which you must adhere in order to change an assembler or HLL user exit.

User exits supported under Language Environment

Language Environment provides two user exit routines, one written in assembler (CEEBXITA), and the other in a Language Environment-conforming language or Fortran (CEEBINT). You can find sample jobs containing these user exits in the SCEESAMP sample library.

The user exits supported by Language Environment are shown in Table 58 on page 373.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type of user exit</th>
<th>When invoked</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEBXITA</td>
<td>Assembler user exit</td>
<td>Enclave initialization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enclave termination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Process termination</td>
</tr>
<tr>
<td>CEEBINT</td>
<td>HLL user exit. CEEBINT can be written in C, C++ (with C linkage), Fortran, PL/I or Language Environment-conforming assembler.</td>
<td>Enclave initialization</td>
</tr>
</tbody>
</table>

When CEEBXITA or CEEBINT is linked with the Language Environment initialization/termination library routines during installation, it functions as an installation-wide user exit. The sample CEEWXIXIT or CEEWDXIT in CEE.SCEESAMP can be used to create and bind(link) your exit with Language Environment initialization/termination routines. When CEEBXITA is linked in your load module, it functions as an application-specific user exit. The application-specific exit is used only when you run that application. The installation-wide assembler user exit is not executed. CEEWXIXIT in CEE.SCEESAMP can be used to assist with creating an application-specific user exit.

When your version of CEEBINT is linked with the Language Environment library routines during installation, this version is automatically used at link-edit time for newly built or relinked applications. A new version of CEEBINT will require you to relink your application.
To use an application-specific user exit, you must explicitly include it at link-edit time in the application load module using an MVS INCLUDE link-edit control statement (see “Using the INCLUDE statement” on page 64 for more information). Any time that the application-specific exit is modified, it must be relinked with the application.

The assembler user exit interface is described in “CEEBXITA assembler user exit interface” on page 378. The HLL user exit interface is described in “CEEBINT high-level language user exit interface” on page 387.

Using the assembler user exit CEEBXITA

CEEBXITA tailors the characteristics of the enclave before its establishment. It must be written in assembler language because an HLL environment is not yet established when the exit is invoked. You cannot code CEEBXITA as an XPLINK application. However, since CEEBXITA is called directly by Language Environment and not by the application, a non-XPLINK CEEBXITA can be statically bound in the same program object with an XPLINK application. CEEBXITA is driven for enclave initialization and enclave termination regardless of whether the enclave is the first enclave in the process or a nested enclave. CEEBXITA can differentiate easily between first and nested enclaves. For more information about nested enclaves, see Chapter 31, “Using nested enclaves,” on page 469.

CEEBXITA is invoked very early during the initialization process, before enclave initialization is complete. The enclave initialization code recognizes runtime options contained in CEEBXITA.

The assembler user exit is supported with POSIX(ON) and in a threaded environment. Within a given Language Environment process, the following functions are driven on the initial thread:

• Initialization of the first enclave within a process
• Termination of the first enclave within a process
• Termination of the process

For nested enclaves, the following functions are driven:

• Nested enclave initialization
• Nested enclave termination

The abend percolation list is applied to all threads in the enclave as specified in the assembler user exit.

Using the HLL initialization exit CEEBINT

CEEBINT is invoked just before the invocation of the application code. Under Language Environment, this exit can be written in C, C++, Fortran, PL/I, or in Language Environment-conforming assembler. When CEEBINT is written in C++, it must be declared as extern “C” in the C++ source. CEEBINT cannot be written in COBOL, even though COBOL applications can use this HLL user exit. You cannot code CEEBINT as an XPLINK application. However, since CEEBINT is called directly by Language Environment and not the application, a non-XPLINK CEEBINT can be statically bound in the same program object with an XPLINK application. When CEEBINT is invoked, the runtime environment is fully operational and all Language Environment-conforming HLLs are supported.

PL/I and C compatibility

The following OS PL/I Version 2 Release 3 user exits are supported for compatibility under Language Environment:

• IBMBXITA (z/OS batch version)
• IBMFXITA (CICS version)

Note: Enterprise PL/I for z/OS does not support the IBMBXITA, IBMFXITA and IBMBINT user exits.

For information about IBMBXITA and IBMBINT, see the appropriate migration guide in IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735) or refer to z/OS XL C/C++ Programming Guide.

Default versions of these user exits are not supplied under Language Environment. Instead, Language Environment supplies a default version of CEEBXITA.
Table 59 on page 375 describes the order of precedence if the IBMBXITA and IBMFXITA user exits are found in the same root load module with CEEBXITA.

<table>
<thead>
<tr>
<th>CEEBXITA present</th>
<th>IBMBXITA present under z/OS batch, IBMFXITA present under CICS</th>
<th>Exit driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Default version of CEEBXITA</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>CEEBXITA</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>IBMBXITA under z/OS batch; IBMFXITA under CICS</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>CEEBXITA</td>
</tr>
</tbody>
</table>

**Using sample assembler user exits**

You can use the sample assembler user exit programs distributed with Language Environment to modify the code for the requirements of your application. Choose a sample program appropriate for your application. The following assembler user exit programs are delivered with Language Environment:

<table>
<thead>
<tr>
<th>Example user exit</th>
<th>Operating system</th>
<th>Where found</th>
<th>Language (if language-specific)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEBXITA</td>
<td>MVS (default)</td>
<td>SCEESAMP</td>
<td></td>
</tr>
<tr>
<td>CEEBXITC</td>
<td>TSO</td>
<td>SCEESAMP</td>
<td></td>
</tr>
<tr>
<td>CEECXITA</td>
<td>CICS (default)</td>
<td>SCEESAMP</td>
<td></td>
</tr>
<tr>
<td>CEEBX05A</td>
<td>MVS</td>
<td>SCEESAMP</td>
<td>VS COBOL II compatibility</td>
</tr>
</tbody>
</table>

If you install Language Environment at your site without modifying it, your system defaults are CEEBXITA and CEECXITA for MVS and CICS. You can find the source code for CEEBXITA, CEEBXITC, CEECXITA, and CEEBX05A on MVS in the sample library SCEESAMP.

The assembler user exit CEEBXITA performs functions for enclave initialization, normal and abnormal enclave termination, and process termination. CEEBXITA must be written in assembler language, because an HLL environment might not be established when the exit is invoked.

You can set up user exits for tasks such as:

- Installation accounting and charge back
- Installation audit controls
- Programming standard enforcement
- Common application runtime support

**When user exits are invoked**

Figure 98 on page 376 shows the timing of the invocations of the user exits at initialization and termination processing.
Figure 98: Location of user exits

In Figure 98 on page 376, runtime user exits are invoked in the following sequence:

1. Assembler user exit is invoked for enclave initialization.
2. Environment is established.
3. HLL user exit is invoked.
4. Main routine is invoked.
5. Main routine returns control to caller.
6. Assembler user exit is invoked for termination of the enclave. CEEBXITA is invoked for enclave termination processing after all application code in the enclave has completed, but before any enclave termination activity.

7. Environment is terminated.

8. Assembler user exit is invoked for termination of the process. CEEBXITA is invoked again when the Language Environment process terminates.

Language Environment provides the CEEBXITA assembler user exit for termination but does not provide a corresponding HLL termination user exit.

CEEBXITA behaves differently, depending upon when it is invoked, as described in the following topics.

**CEEBXITA behavior during enclave initialization**

The CEEBXITA assembler user exit is invoked before enclave initialization is performed. You can use CEEBXITA to help establish your application runtime environment. For example, in the assembler user exit you can specify the stack and heap runtime options and allocate data sets. You can also use the user exit to interrogate program parameters supplied in the JCL and change them if you want. In addition, you can specify runtime options in the user exit by using the CEEAUE_A_OPTIONS field of the assembler interface.

CEEBXITA returns control to Language Environment initialization.

**CEEBXITA behavior during enclave termination**

The CEEBXITA assembler exit is invoked after the user code for the enclave has completed, but before the occurrence of any enclave termination activity. In other words, the assembler user exit for termination is invoked when the environment is still active. For example, CEEBXITA is invoked before the storage report is produced (if you requested one), data sets are closed, and CODE is invoked for enclave termination.

The assembler user exit permits you to request an abend. You can also request a dump to assist in problem diagnosis. Because termination activities have not yet begun when the user exit is invoked, the majority of storage has not been modified when the dump is produced.

You can request the abend and dump in the assembler user exit for all enclave-terminating events including:

- The situation that occurs in PL/I when the ON condition (including ERROR or FINISH) is raised and one of the following conditions is true:
  - The program does not have an appropriate ON-unit.
  - The ON-unit does not terminate with a GOTO.
  - The GOTO is not allowed.

  This rule applies only to the conditions that cause termination of the program.

- Return from the main routine
- A debug tool QUIT command
- An HLL stop statement such as:
  - C exit()
  - COBOL STOP RUN
  - PL/I STOP or EXIT
  - Fortran STOP

- An unhandled condition of severity 2 or above

If a dump is requested in the user assembler exit and an unhandled condition has occurred, this dump will overwrite the dump taken by TERMTHDACT(UADUMP).
CEEBXITA behavior during process termination

The CEEBXITA assembler exit is invoked after:

- All enclaves have been terminated,
- The enclave resources have been relinquished,
- Any files managed by Language Environment have been closed,
- IBM z/OS Debugger has been terminated,

At this time you can free allocated files and request an abend.

During termination, CEEBXITA can interrogate the Language Environment reason and return codes and, if necessary, request an abend with or without a dump. This can be done at either enclave or process termination.

Specifying abend codes to be percolated by Language Environment

The assembler user exit, when invoked for initialization, might return a list of abend codes (contained in the CEEAUE_A_AB_CODES field of the assembler user exit interface—see “CEEBXITA assembler user exit interface” on page 378) that are to be percolated by Language Environment.

On non-CICS systems, this list is contained in the CEEAUE_A_AB_CODES field of the assembler user exit interface. (See “CEEBXITA assembler user exit interface” on page 378.) Both system abends and user abends can be specified in this list. The abend percolation list specified in the assembler user exit applies to all threads in the enclave.

When TRAP(ON) is in effect, and the abend code is in the CEEAUE_A_AB_CODES list, Language Environment percolates the abend. Normal Language Environment condition handling is never invoked to handle these abends. This feature is useful when you do not want Language Environment condition handling to intervene for certain abends, such as when IMS issues a user ABEND code 777.

When TRAP(OFF) is specified and there is a program interrupt, the user exit for termination is not driven.

Actions taken for errors that occur within the exit

If any errors occur during the enclave initialization user exit, the standard system action occurs because Language Environment condition handling has not yet been established.

Any errors occurring during the enclave termination user exit lead to abnormal termination (through an abend) of the Language Environment environment.

If there is a program check during the enclave termination user exit and TRAP(ON) is in effect, the application ends abnormally with ABEND code 4044 and reason code 2. If there is a program check during the enclave termination user exit and TRAP(OFF) has been specified, the application ends abnormally without additional error checking support. Language Environment performs no condition handling; error handling is performed by the operating system.

Language Environment takes the same actions as described above for program checks during the process termination user exit.

CEEBXITA assembler user exit interface

You can modify CEEBXITA to perform any function you need, but the exit must have the following attributes after you modify it at installation:

- The user-supplied exit must be named CEEBXITA.
- The exit must be reentrant.
- The exit must be capable of executing in AMODE(ANY) and RMODE(ANY).
Guidelines for using CEEBXITA

Installation-wide:

- You must bind (link) the exit with the appropriate Language Environment initialization/termination routines after modification.
- Use the sample customization jobs CEEWDXIT and CEEWCXIT to assist with creating and binding (linking) your exit with Language Environment initialization/termination routines.
- See the section on changing the assembler language user exit in *z/OS Language Environment Customization* for more details.

Application-specific:

- You must bind (link) the exit with your application.
- Use the sample job CEEWUXIT to assist with creating your exit.
- See the section on changing the assembler language user exits in *z/OS Language Environment Customization* for more details.

If a user exit is modified, you are responsible for conforming to the interface shown in Figure 99 on page 380. This user exit must be written in assembler. You cannot code CEEBXITA as an XPLINK application. However, since CEEBXITA is called directly by Language Environment and not the application, a non-XPLINK CEEBXITA can be statically bound in the same program object with an XPLINK application.
When the user exit is called, register 1 points to a word that contains the address of the CEEAUE control block. The high-order bit is on.

The CEEAUE control block contains the following fullwords:

**CEEAUE_LEN (input parameter)**
A fullword integer that specifies the total length of this control block. For Language Environment, the length is 48 bytes.

**CEEAUE_FUNC (input parameter)**
A fullword integer that specifies the function code. Language Environment supports the following function codes:

1  
Initialization of the first enclave within a process.
Termination of the first enclave within a process.

Nested enclave initialization.

Nested enclave termination.

Process termination.

The user exit should ignore function codes other than those numbered from 1 through 5.

**CEEAUE.RETC (input/output parameter)**

A fullword integer that specifies the return or abend code. CEEAUE.RETC has different meanings, depending on CEEAUE.ABND:

- If the flag CEEAUE.ABND (see below) is off, this fullword is interpreted as the Language Environment return code placed in register 15.
- If the flag CEEAUE.ABND is on, CEEAUE.RETC is interpreted as an abend code used when an abend is issued. (This could be either an EXEC CICS ABEND or an SVC13.)

**CEEAUE.RSNC (input/output parameter)**

A fullword integer that specifies the reason code for CEEAUE.RETC:

- If the flag CEEAUE.ABND (see below) is off, this word is interpreted as the Language Environment reason code placed in register 0.
- If the flag CEEAUE.ABND is on, CEEAUE.RETC is interpreted as an abend reason code used when an abend is issued.

This field is ignored when an EXEC CICS ABEND is issued.

**CEEAUE_FLAGS**

Contains four 1-byte flags. CEEBXITA uses only the first byte but reserves the remaining flags. All unspecified bits and bytes must be 0. The layout of these flags is shown in Figure 100 on page 381:

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>0... .... Normal termination</th>
<th>1... .... Abnormal termination</th>
<th>.x.. .... CEEAUE.ABND</th>
<th>.0... .... Terminate with CEEAUE.RETC</th>
<th>.1... .... ABEND with CEEAUE.RETC and CEEAUE.RSNC given</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.x... .... CEEAUE.DUMP</td>
<td>.0... .... If CEEAUE.ABND=0, ABEND with no dump</td>
<td>.1... .... If CEEAUE.ABND=1, ABEND with a dump</td>
<td>.x... .... CEEAUE.STEPS</td>
<td>...0 ... ABEND the task</td>
</tr>
<tr>
<td>Byte 1</td>
<td>0000 0000 Reserved for future use</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 2</td>
<td>0000 0000 Reserved for future use</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 3</td>
<td>0000 0000 Reserved for future use</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 100: CEEAUE_FLAGS format**

Byte 0 (CEEAUE_FLAG1) has the following meaning:

**CEEAUE_ABTERM (input parameter)**
OFF Indicates that the enclave is terminating normally (severity 0 or 1 condition).

ON Indicates that the enclave is terminating with an Language Environment return code modifier of 2 or greater. This could, for example, indicate that a severity 2 or greater condition was raised but not handled.

**CEEAUE_ABND** (input/output parameter)

OFF Indicates that the enclave should terminate without an abend being issued. Thus, CEEAUE_RETC and CEEAUE_RSNC are placed into register 15 and register 0 and returned to the enclave creator.

ON Indicates that the enclave terminates with an abend. Thus, CEEAUE_RETC and CEEAUE_RSNC are used by Language Environment in the invocation of the abend. During running in CICS, an EXEC CICS ABEND command is issued.

The TRAP runtime option does not affect the setting of CEEAUE_ABND.

When the ABTERMENC(ABEND) runtime option is specified, the enclave always terminates with an abend when there is an unhandled condition of severity 2 or greater, regardless of the setting of the CEEAUE_ABND flag. See “Termination behavior for unhandled conditions” on page 135 for a detailed explanation of how the CEEAUE_ABND parameter can affect the behavior of the ABTERMENC runtime option.

**CEEAUE_DUMP** (output parameter)

OFF Indicates that when you request an abend, an abend is issued without requesting a dump.

ON Indicates that when you request an abend, an abend requesting a dump is issued.

**CEEAUE_STEPS** (output parameter)

OFF Indicates that when you request an abend, an abend is issued to abend the entire TASK.

ON Indicates that when you request an abend, an abend is issued to abend the STEP.

This parameter is ignored under CICS.

**CEEAUE_A_CC_PLIST** (input/output parameter)

A fullword pointer to the parameter address list of the application program.

If the parameter is not a character string, CEEAUE_A_CC_PLIST contains the register 1 value as passed by the calling program or operating system at the time of program entry.

If the parameter inbound to the MAIN routine is a character string, CEEAUE_A_CC_PLIST contains the address of a fullword address that points to a halfword prefixed string. If this string is altered by the user exit, the string must not be extended in place.

**CEEAUE_A_WORK** (input parameter)

A fullword pointer to a 256-byte work area that the exit can use. On entry it contains binary zeros and is doubleword-aligned.

This area does not persist across exits.

**CEEAUE_A_OPTIONS** (output parameter)

Upon return, this field contains a fullword pointer to the address of a halfword-length prefixed character string that contains runtime options. These options are honored only during the initialization of an enclave. When invoked for enclave termination, this field is ignored.
These runtime options override all other sources of runtime options except those that are specified as NONOVR.

Under CICS, the STACK runtime option cannot be modified with the assembler user exit.

**CEEAUE_USERWD (input/output parameter)**
A fullword whose value is maintained without alteration and passed to every user exit. Upon entry to the enclave initialization user exit, it is zero. Thereafter, the value of the user word is not altered by Language Environment or any member libraries. The user exit might change the value of this field, and Language Environment maintains that value. This allows the user exit to acquire a work area, initialize it, and pass it to subsequent user exits. The work area might be freed by the termination user exit.

**CEEAUE_A_AB_CODES (output parameter)**
During the initialization exit, this field contains a fullword address of a table of abend codes that the Language Environment condition handler percolates while in the (E)STAE exit. Therefore, the application does not have the chance to address the abend. This table is honored before shunt routines. The table consists of:

- A fullword count of the number of abend codes that are to be percolated
- A fullword for each of the particular abend codes that are to be percolated

The abend codes might be either user abend codes or system abend codes. User abend codes are specified by F'uuu'. For example, if you want to percolate user ABEND 777, a F'777' would be coded. System abend codes are specified by X'00sss000'.

This parameter is not enabled under CICS.

**CEEAUE_FBCODE (input parameter)**
Contains a fullword address of the condition token with which the enclave terminated. If the enclave terminates normally (that is, not due to a condition), the condition token is zero.

**CEEAUE_PAGE (input parameter)**
This parameter indicates whether PL/I BASED variables that are allocated storage outside of AREAs are allocated on a 4K-page boundary. You can specify in the field the minimum number of bytes of storage that must be allocated. Your allocation request must be an exact multiple of 4 KB.

The IBM-supplied default setting for CEEAUE_PAGE is 32768 (32 KB).

If CEEAUE_PAGE is set to zero, PL/I BASED variables can be placed on other than 4K-page boundaries.

CEEAUE_PAGE is honored only during enclave initialization, that is, when CEEAUE_FUNC is 1 or 3.

The offset of CEEAUE_PAGE under Language Environment is different than under OS PL/I Version 2 Release 3.

**Parameter values in the assembler user exit**
The parameters described in “CEEBXITA assembler user exit interface” on page 378 contain different values depending on how the user exit is used. Table 61 on page 384 and Table 62 on page 386 describe the possible values for the parameters based on how the assembler user exit is invoked.
Table 61: Parameter values in the assembler user exit (Part 1). The assembler user exit contains these parameter values depending on when it is invoked.

<table>
<thead>
<tr>
<th>When invoked</th>
<th>CEEAUE_LEN</th>
<th>CEEAUE_RETC</th>
<th>CEEAUE_RSNC</th>
<th>CEEAUE_FLAGS</th>
<th>CEEAUE_A_CC_PLIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>First enclave within process initialization:</td>
<td>48</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Entry CEEAUE_FUNC = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Upon entry, CEEAUE_A_CC_PLIST contains the register 1 value from the operating system. It contains the user parameters. You can alter it in a user exit. Upon return, the CEEAUE_A_CC_PLIST is processed and merged as the invocation string.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First enclave within process initialization:</td>
<td>0, or abend code if CEEAUE_ABND = 1</td>
<td>0, or reason code for CEEAUE_RETC if CEEAUE_ABND = 1</td>
<td></td>
<td></td>
<td>See Note “1” on page 386.</td>
</tr>
<tr>
<td>Return</td>
<td>Register 1, used as the new parameter list. CEEAUE_A_CC_PLIST contains the user parameters. You can alter it in a user exit. Upon return, the CEEAUE_A_CC_PLIST is processed and merged as the invocation string.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First enclave within process termination:</td>
<td>48</td>
<td>Return code issued by application that is terminating.</td>
<td>Reason code that accompanies CEEAUE_RETC.</td>
<td></td>
<td>See Note “2” on page 386.</td>
</tr>
<tr>
<td>Entry CEEAUE_FUNC = 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If CEEAUE_ABND = 0, the return code placed into register 15 when the enclave terminates.</td>
<td>If CEEAUE_ABND = 0, the enclave reason code.</td>
<td></td>
<td></td>
<td>See Note “1” on page 386.</td>
</tr>
<tr>
<td>Return</td>
<td>If CEEAUE_ABND = 1, the abend code.</td>
<td>If CEEAUE_ABND = 1, the abend reason code.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 61: Parameter values in the assembler user exit (Part 1). The assembler user exit contains these parameter values depending on when it is invoked. (continued)

<table>
<thead>
<tr>
<th>When invoked</th>
<th>CEEAU_LEN</th>
<th>CEEAU_RETC</th>
<th>CEEAU_RSNC</th>
<th>CEEAU_FLAGS</th>
<th>CEEAU_A_CC_PLIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested enclave initialization: Entry</td>
<td>48</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>The register 1 value discovered in a nested enclave creation. CEEAU_A_CC_PLIST contains the user parameters. You can alter it in a user exit. Upon return, the CEEAU_A_CC_PLIST is processed and merged as the invocation string.</td>
</tr>
<tr>
<td>CEEAU_FUNC = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested enclave initialization: Return</td>
<td>0, or if CEEAU_ABND = 1, the abend code.</td>
<td>0, or if CEEAU_ABND = 1, reason code for CEEAU_RETC.</td>
<td>See Note “1” on page 386.</td>
<td>Register 1 used as the new enclave parameter list. CEEAU_A_CC_PLIST contains the user parameters. You can alter it in a user exit. Upon return, the CEEAU_A_CC_PLIST is processed and merged as the invocation string.</td>
<td></td>
</tr>
<tr>
<td>Nested enclave termination: Entry</td>
<td>48</td>
<td>Return code issued by enclave that is terminating.</td>
<td>Reason code accompanying CEEAU_RETC.</td>
<td>See Note “2” on page 386.</td>
<td></td>
</tr>
<tr>
<td>CEEAU_FUNC = 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested enclave termination: Return</td>
<td>If CEEAU_ABND = 0, the return code from the enclave.</td>
<td>If CEEAU_ABND = 0, the enclave reason code.</td>
<td>See Note “1” on page 386.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If CEEAU_ABND = 1, the abend code.</td>
<td>If CEEAU_ABND = 1, the enclave reason code.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process termination: Entry</td>
<td>48</td>
<td>Return code presented to the invoking system in register 15 that reflects the value returned from the &quot;first enclave within process termination&quot;.</td>
<td>Reason code accompanying CEEAU_RETC that is presented to the invoking system in register 0 and reflects the value returned from the &quot;first enclave within process termination&quot;.</td>
<td>See Note “3” on page 386.</td>
<td></td>
</tr>
<tr>
<td>Function code = 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 61: Parameter values in the assembler user exit (Part 1). The assembler user exit contains these parameter values depending on when it is invoked. (continued)

<table>
<thead>
<tr>
<th>When invoked</th>
<th>CEEAUE_LEN</th>
<th>CEEAUE_RETC</th>
<th>CEEAUE_RSNC</th>
<th>CEEAUE_FLAGS</th>
<th>CEEAUE_A_CC_PLIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process termination</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>If CEEAUE_ABND = 0, return code from the process.</td>
<td>If CEEAUE_ABND = 0, the reason code for CEEAUE_RETC from the process.</td>
<td>See Note “1” on page 386.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. **CEEAE_FLAGS:**

   - CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing
   - CEEAUE_DUMP = 1 if the abend should request a dump
   - CEEAUE_STEPS = 1 if the abend should abend the step
   - CEEAUE_STEPS = 0 if the abend should abend the task

2. **CEEAE_FLAGS:**

   - CEEAUE_ABTERM = 1 if the application is terminating with an Language Environment return code modifier of 2 or greater, or 0 otherwise
   - CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing
   - CEEAUE_DUMP = 0
   - CEEAUE_STEPS = 0

3. **CEEAE_FLAGS:**

   - CEEAUE_ABTERM = 1 if the last enclave is terminating abnormally (that is, a Language Environment return code modifier is 2 or greater). This reflects the value returned from the "first enclave within process termination".
   - CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing "first enclave within process termination" (function code 2).
   - CEEAUE_DUMP = 0
   - CEEAUE_STEPS = 0

Table 62: Parameter values in the assembler user exit (Part 2). The assembler user exit contains these parameter values depending on when it is invoked.

<table>
<thead>
<tr>
<th>When invoked</th>
<th>CEEAUE_A_WORK</th>
<th>CEEAUE_A_OPTIONS</th>
<th>CEEAUE_USERWD</th>
<th>CEEAUE_A_AB_CODES</th>
<th>CEEAUE_FBCODE</th>
<th>CEEAUE_PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>First enclave within process initialization:</td>
<td>Address of a 256-byte work area of binary zeros.</td>
<td>0</td>
<td>0</td>
<td>Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEEAUE_FUNC = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First enclave within process initialization:</td>
<td>Pointer to address of a halfword prefixed character string containing runtime options, or 0.</td>
<td>The value of CEEAUE_USERWD for all subsequent exits.</td>
<td>Pointer to the abend codes table, or 0.</td>
<td>User specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 62: Parameter values in the assembler user exit (Part 2). The assembler user exit contains these parameter values depending on when it is invoked. (continued)

<table>
<thead>
<tr>
<th>When invoked</th>
<th>CEEAUE_A_WORK</th>
<th>CEEAUE_A_OPTIONS</th>
<th>CEEAUE_USERWD</th>
<th>CEEAUE_A_AB_CODES</th>
<th>CEEAUE_FBCODE</th>
<th>CEEAUE_PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>First enclave within</td>
<td>Address of a 256-byte area of binary zeros.</td>
<td>Return value from previous exit.</td>
<td>Feedback code causing termination.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>process termination:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry CEEAUE_FUNC = 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First enclave within</td>
<td>The value of</td>
<td>Return value from</td>
<td>Minimum number of</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>process termination:</td>
<td>CEEAUE_USERWD</td>
<td>previous exit.</td>
<td>storage bytes to</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>for all subsequent</td>
<td></td>
<td>be allocated for</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>exits.</td>
<td></td>
<td>PL/I BASED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested enclave initialization:</td>
<td>Address of a 256-byte work area of binary zeros.</td>
<td>Pointer to fullword address that points to a halfword prefixed length string containing runtime options, or 0.</td>
<td>The value of CEEAUE_USERWD for all subsequent exits.</td>
<td>Pointer to abend codes table, or 0.</td>
<td>User specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).</td>
<td></td>
</tr>
<tr>
<td>Entry CEEAUE_FUNC = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested enclave termination:</td>
<td>The value of</td>
<td>Return value from</td>
<td>Feedback code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>— Entry CEEAUE_FUNC = 4</td>
<td>CEEAUE_USERWD</td>
<td>previous exit.</td>
<td>causing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>for all subsequent</td>
<td></td>
<td>termination.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested enclave termination:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>The value of CEEAUE_USERWD for all subsequent exits.</td>
<td></td>
<td>Feedback code causing termination.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested enclave termination:</td>
<td>Address of a 256-byte work area of binary zeros.</td>
<td>Return value from previous exit.</td>
<td>Feedback code causing termination.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>The value of CEEAUE_USERWD for all subsequent exits.</td>
<td></td>
<td>Feedback code causing termination.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CEEINT high-level language user exit interface

Language Environment provides CEEINT for enclave initialization. You can code CEEINT in non-XPLINK C and C++, Fortran, PL/I, or Language Environment-conforming assembler. You cannot code CEEINT as an XPLINK application. CEEINT is not invoked for an XPLINK application. COBOL programs can use CEEINT, but CEEINT cannot be written in COBOL or be used to call COBOL programs.

CEEINT is supported with POSIX(ON) and in a threaded environment. It is driven only on the initial thread.

You can modify CEEINT to perform any function desired, although the exit must have the following attributes after you modify it:

- The user exit must not be a main-designated routine. That is, it must not be a C or C/++ main function, and OPTIONS(MAIN) must not be specified for PL/I applications.
- CEEINT must be linked with compiled code. If you do not provide an initialization user exit, an IBM-supplied default, which simply returns control to your application, is linked with the compiled code. When written in C/++, CEEINT must be linked with your application and it can only function as an application-specific user exit.
- The exit cannot be written in COBOL.
When CEEBINT is written in C/C+++, the following must be coded so that SMP/E can maintain the CSECT and properly link the intended user exit:

```c
#pragma map(CEEBINT, "CEEBINT")
```

The exit should be coded so that it returns for all unknown function codes.

C or C/++ constructs such as the `exit()`, `abort()`, `raise(SIGTERM)`, and `raise(SIGABRT)` functions terminate the enclave.

A PL/I EXIT or STOP statement terminates the enclave.

Use the callable service IBMHKS to turn hooks on and off. For more information about IBMHKS, see PL/I for MVS & VM Programming Guide.

C or C/++ functions such as `exit()`, `abort()`, `raise(SIGTERM)`, and `raise(SIGABRT)` terminate the entire application as well as the user exit.

CEEBINT is invoked after the enclave has been established, after the IBM z/OS Debugger initial command string has been processed, and before the invocation of compiled code. When invoked, it is passed a parameter list. The parameters are all fullwords and are defined as:

**Number of arguments in parameter list (input)**
- A fullword binary integer
  - On entry: Contains 7
  - On exit: Not applicable

**Return code (output)**
- A fullword binary integer
  - On entry: 0
  - On exit: Able to be set by the exit, but not interrogated by Language Environment

**Reason code (output)**
- A fullword binary integer
  - On entry: 0
  - On exit: Able to be set by the exit, but not interrogated by Language Environment

**Function code (input)**
- A fullword binary integer
  - On entry: 1, indicating the exit is being driven for initialization
  - On exit: Not applicable

**User word (input/output)**
- A fullword binary integer
  - On entry: Value of the user word (CEEAUE_USERWD) as set by the assembler user exit.
  - On exit: The value set by the user exit, maintained by Language Environment and passed to subsequent user exits. It can be accessed from the `main()` function through the system programming facilities C `__xusr()` function.

**Address of the main program entry point (input)**
- A fullword binary address
  - On entry: The address of the routine that gains control first
  - On exit: Not applicable

**Exit List Address (output)**
- The address of the exit list control block, Exit_list
  - On entry: 0
- On exit: 0, unless you establish a hook exit, in which case you would set this pointer and fill in the relevant control blocks. The control blocks for Exit_list and Hook_exit are shown in the following figure.

As supplied, CEEBINT has only one exit defined that you can establish — the hook exit that is described by the Hook_exit control block. This exit gains control when hooks that are generated by the PL/I compiler TEST option are executed. You can establish this exit by setting appropriate pointers (A_Exits to Exit_list to Hook_exit).

**Figure 101: Exit_list and hook_exit control blocks**

The control block Exit_list exit contains the following fields:

**Exit_list_len**
- The length of the control block; it must be 1
Exit_list_hooks
The address of the Hook_exit control block

The control block for the hook exit must contain the following fields:

**Hook_exit_len**
The length of the control block

**Hook_exit_rtn**
The address of a routine you want invoked for the exit. When the routine is invoked, it is passed the address of this control block. Since this routine is invoked only if the address you specify is nonzero, you can turn the exit on and off.

**Hook_exit_fnccode**
The function code with which the exit is invoked. This is always 1.

**Hook_exit_retcode**
The return code set by the exit. You must ensure that it conforms to the following specifications:

- 0 Requests that the IBM z/OS Debugger be invoked next.
- 4 Requests that the program resume immediately.
- 16 Requests that the program be terminated.

**Hook_exit_rsncode**
The reason code set by the exit. This is always zero.

**Hook_exit_userwd**
The user word passed to the user exits CEEBXITA and CEEBINT

**Hook_exit_ptr**
An exit-specific user word

**Hook_exit_reserved**
Reserved

**Hook_exit_dsa**
The contents of register 13 when the hook was executed

**Hook_exit_addr**
The address of the hook instruction executed
Chapter 29. Assembler considerations

You can run applications written in assembler language in Language Environment. Applications written in Language Environment-conforming HLLs can also call or be called by assembler language applications. It is important to note that Fortran applications cannot call CEEHDLR or any other Language Environment callable service directly, therefore Fortran condition handling must be done by calling an assembler application to provide condition handling support. Most CEE services can be called from Fortran using AFHCEEN or AFHCEEF, as described in *z/OS Language Environment Runtime Application Migration Guide*.

This topic discusses considerations for assembler applications and introduces library routine retention, a function that can provide performance improvement for applications or subsystems running on z/OS.

You can write assembler language applications that conform to the XPLINK call linkage. The *z/OS XL C/C++ Programming Guide* describes how to create XPLINK assembler applications using the EDCXPRLG, EDCXEPPLG, and EDCXCALL macros, and describes the XPLINK register conventions, parameter passing conventions and stack layout. *z/OS Language Environment Vendor Interfaces* has details on the XPLINK architecture that will be useful to an assembler programmer.

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 runtime environment, your assembler application must preserve the use of certain registers and storage areas in a consistent way. Calling conventions for non-XPLINK assembler programs must follow the standard S/370 linkage conventions. Calling conventions for XPLINK assembler programs must follow the XPLINK linkage conventions. In addition, your assembler program is restricted from using some operating system services. These conventions and restrictions are described in this section.

Compatibility considerations

If you are coding a new assembler routine that you want to conform to the Language Environment interface 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 399. *Language Environment-conforming assembler routine* refers to an assembler routine coded using the CEEENTRY and associated macros.

Control blocks

Assembler routines that rely on control blocks that were versions of C, COBOL, Fortran, and PL/I (for example, routines that check flags or switches in these control blocks) might not run under Language Environment. These control blocks might have changed.

Save areas

Any non-XPLINK assembler routine used within the scope of a Language Environment application must use standard S/370 save area conventions. Any XPLINK assembler routine used within the scope of a Language Environment application must use XPLINK save area conventions.

Note:

1. To call a COBOL program from assembler, set the first two bytes of the save area to hex zero.
2. Language Environment does not support the linkage stack.
3. A non-Language Environment-conforming assembler routine must have its own save area.
**CICS**

When running with a release of CICS TS earlier than CICS TS 3.1, Language Environment-conforming assembler **main** routines are not supported under CICS.

**C and Fortran duplicate names**

Several external names, shown in column one of Table 63 on page 392, are identical in C and Fortran. If any of the names is used in an assembler program as an external reference, the C—not the Fortran—entity is obtained. If you wish to obtain the Fortran version, you can instead reassemble using the names shown in column two of Table 63 on page 392 as a substitute for the C names in column one.

For example, if your assembler program currently references the Fortran ABS function with the instruction:

```
ABSADDR DC V(ABS)
```

you could instead reassemble it with the instruction:

```
ABSADDR DC V(A#ABS)
```

to obtain the Fortran function as before. The C versions of the functions might additionally require a different parameter-list format.

As an alternative to changing the conflicting names in an assembler routine and then reassembling, you can relink the existing routine following the procedure explained in “Resolving library module name conflicts between Fortran and C” on page 15.

| **Table 63: C external names and their analogous Fortran names** |
|---------------|---------------|
| **C external name** | **Fortran external name** |
| ABS | A#ABS |
| ACOS | A#COS |
| ASIN | A#SIN |
| ATAN | A#TAN |
| ATAN2 | A#TAN2 |
| CLOCK | CLOCK# |
| COS | C#OS |
| COSH | C#OSH |
| ERF | E#RF |
| ERFC | E#RFC |
| EXIT | EXIT# |
| EXP | E#XP |
| GAMMA | G#AMMA |
| LOG | A#LOG |
| LOG10 | A#LOG1 |
| SIN | S#IN |
| SINH | S#INH |
| SQRT | S#QRT |
| TAN | T#AN |
Table 63: C external names and their analogous Fortran names (continued)

<table>
<thead>
<tr>
<th>C external name</th>
<th>Fortran external name</th>
</tr>
</thead>
<tbody>
<tr>
<td>TANH</td>
<td>T#ANH</td>
</tr>
</tbody>
</table>

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 and non-Language Environment-conforming assembler each has its own requirements for register conventions when running under Language Environment.

Language Environment-conforming assembler

When you use the macros listed in “Assembler macros” on page 399 to write your Language Environment-conforming assembler routines, the macros generate code that follows the required register conventions.

On entry into the Language Environment-conforming non-XPLINK assembler main routine, registers must contain the following values because they are passed without change to the CEEENTRY macro:

- **R0**: Undefined
- **R1**: Address of the parameter list, or zero if no parameters are passed
- **R13**: Caller's standard register save area
- **R14**: Return address
- **R15**: Entry point address

On entry into the Language Environment-conforming assembler subroutine, these registers must contain the following values when NAB=YES is specified on the CEEENTRY macro:

- **R0**: Reserved
- **R1**: Address of the parameter list, or zero
- **R12**: Common anchor area (CAA) address
- **R13**: Caller's DSA
- **R14**: Return address
- **R15**: Entry point address

**All others**: Undefined

On entry into a Language Environment-conforming assembler routine, CEEENTRY loads the caller's registers (R14 through R12) in the DSA provided by the caller. After it allocates a DSA (which sets the NAB field correctly in the new DSA), the first halfword of the DSA is set to hex zero and the backchain is set properly.

At all times while the Language Environment-conforming non-XPLINK assembler routine is running, R13 must point to the routine's DSA.
Assembler considerations

At call points, R12 must contain the common anchor area (CAA) address, except in the following cases:
- When calling a COBOL program
- When calling an assembler routine that is not Language Environment-conforming
- When calling a Language Environment-conforming assembler routine that specifies NAB=NO on the CEEENTRY macro

On exit from a Language Environment-conforming assembler routine, these registers contain:

- **R0**: Undefined
- **R1**: Undefined
- **R14**: Undefined
- **R15**: Undefined
- **All others**: The contents they had upon entry

Non-Language Environment conforming assembler routines

When you run a non-Language Environment-conforming routine in Language Environment, you must observe the following conventions:
- R13 must contain the address of the executing routine's own register save area
- The register save area back chain must be set to a valid 31-bit address (if the address is a 24 bit address, the first byte of the address must be hex zeros)
- The first two bytes of the register save area must be hex zeros

Considerations for coding or running assembler routines

This topic summarizes some areas you might need to consider when coding or running an assembler routine under Language Environment.

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 routine, or in a user COBOL routine compiled with the THREAD compiler option
- Just before a return to a C routine or to a return to a user COBOL routine compiled with the THREAD compiler option
- Just before an invocation of a Language Environment library from a user routine

C routines or COBOL routines compiled with the THREAD compiler option 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 CEEHDLR (see z/OS Language Environment Programming Reference for more information about CEEHDLR).

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 standardized form of the inbound parameter list for the assembler main routine any time after routine initialization by using one of the following:

- The CEE3PRM and CEE3PR2(query parameter string) callable service described in z/OS Language Environment Programming Reference.

  What CEE3PRM and CEE3PR2 return depends on the operating system you run under, and the runtime or compiler options you specify. See “What the enclave returns from CEE3PRM and CEE3PR2” on page 477 for more information.

- The PARMREG output value from the CEEENTRY macro described in “CEEENTRY macro—Generate a Language-Environment-conforming prolog” on page 400.

Overlay programs

Language Environment does not provide explicit support for overlay programs. If programs are overlaid, Language Environment imposes the following restrictions:

- All Language Environment routines and static data must be placed in the root segment.
- All named routines and static data referred to by Language Environment must be in the root segment.
- All ENTRY values or static data addresses passed to any Language Environment service must point to routines in the root segment.
- All routines in the save area chain must be in storage for the whole time that they are in the chain.
- All calls must be inclusive, not exclusive. See your Linkage Editor and Loader User’s Guide for the definitions of these terms.
- Calls that cause a new overlay segment to be loaded must be between two routines in the same language (that is, they cannot be ILC calls).

CEESTART, CEEMAIN, and CEEFMAIN

Assembler programs cannot call or use directly CEESTART, CEEMAIN, or CEEFMAIN as a standard entry point. Results are unpredictable if this rule is violated.

When link-editing an application it must be possible for the link-editor to resolve CEESTART. As long as the NCAL link-editor option is not specified, CEESTART will be automatically resolved. If NCAL is used it becomes necessary to explicitly include CEESTART in the link-edit process.

Mode considerations

The CEEENTRY macro automatically sets the module to AMODE ANY and RMODE ANY. Therefore, when converting to Language Environment-conforming assembler, if data management macros had been coded using 24-bit addressing mode, they should be changed to use 31-bit addressing mode (see z/OS DFSMS Macro Instructions for Data Sets). If it is not possible to change all the modules making up the program to use 31-bit addressing mode (and none of the modules are already set to RMODE 24), then it will be necessary to use the RMODE=24 CEEENTRY option. Alternatively, the module can be set to RMODE 24 during the link-edit process. This is done by specifying the link-edit RMODE option on the invocation PARM or the SETOPT control statement.

Language Environment library routine retention (LRR)

Language Environment library routine retention is a function that provides a performance improvement for those applications or subsystems with the following attributes:

- The application or subsystem invokes programs that require Language Environment.
- The application or subsystem is not Language Environment-conforming. That is, Language Environment is not already initialized when the application or subsystem invokes programs that require Language Environment.
• The application or subsystem, while running under the same task, repeatedly invokes programs that require Language Environment.
• The application or subsystem is not using Language Environment preinitialization services.

Restriction: Language Environment library routine retention is not supported to run on CICS.

The use of library routine retention does not affect the behavior of applications other than improving their performance.

Language Environment provides a macro called CEELRR, which is used in an assembler program to initialize library routine retention and to terminate library routine retention. See “CEELRR macro — Initialize or terminate Language Environment library routine retention” on page 397 for details about the CEELRR macro.

In addition, Language Environment provides three sample programs that use the CEELRR macro:

CEELRRIN
This routine uses the CEELRR macro to initialize a library routine retention environment that does not permit XPLINK applications. The source for this routine can be found in member CEELRRIN in SCEESAMP. The load module associated with this routine can be found in SCEERUN with member name CEELRRIN.

CEELRRXP
This routine uses the CEELRR macro to initialize a library routine retention environment that permits XPLINK applications. The source for this routine can be found in member CEELRRXP in SCEESAMP. The load module associated with this routine can be found in SCEERUN with member name CEELRRXP.

CEELRRTR
This routine uses the CEELRR macro to terminate library routine retention. The source for this routine can be found in member CEELRRTR in SCEESAMP. The load module associated with this routine can be found in SCEERUN with member name CEELRRTR.

When library routine retention has been initialized, Language Environment keeps a subset of its resources in memory after the environment terminates. As a result, subsequent invocations of programs in the same task that caused Language Environment to be initialized are much faster because the resources can be reused without having to be reacquired and reinitialized.

When library routine retention has been initialized, the resources that Language Environment keeps in memory when it terminates include the following:
• Language Environment runtime load modules
• Language Environment storage associated with the management of the runtime load modules
• Language Environment storage for startup control blocks

When library routine retention is terminated, the resources that Language Environment kept in memory are freed. (Library routines are deleted and storage is freed.)

Note:
1. If library routine retention is initialized, and the task in which it is being used is terminated, the operating system frees the Language Environment resources as part of task termination.
2. 31-bit XPLINK applications are supported under the LRR environment.

Using library routine retention

If you are going to use library routine retention, you need to be aware of the following:
• Library routine retention cannot be used on CICS.
• To successfully initialize library routine retention or terminate library routine retention, Language Environment must not be currently initialized.
For example, if you use CEELRR with ACTION=INIT in a Language Environment-conforming assembler program, library routine retention is not initialized, because the invocation of the assembler program caused Language Environment to be initialized.

Library routine retention and preinitialization

The Language Environment preinitialization services can be used while library routine retention is initialized. However, the Language Environment resources initialized and terminated with Language Environment preinitialization services are not kept in memory when while library routine retention is initialized. There is no sharing of resources between Language Environment when initialized with preinitialization services and an environment initialized by invoking an HLL program without using preinitialization services. There is no performance benefit of library routine retention for those applications and subsystems that bring up a Language Environment preinitialized environment and then use the preinitialization services to invoke programs that require Language Environment.

CEELRR macro — Initialize or terminate Language Environment library routine retention

CEELRR is used to tell Language Environment to initialize and terminate library routine retention. The macro generates reentrant code.

**label**
Assembler label on this macro generation.

**ACTION=**
The action to be performed by Language Environment with regard to library routine retention. Valid values are INIT and TERM. A value of INIT tells Language Environment to initialize library routine retention. A value of TERM tells Language Environment to terminate library routine retention. You must specify the ACTION value.

**XPLINK=**
The XPLINK keyword allows the application to specify whether XPLINK applications are permitted under the LRR environment. Valid values are YES and NO. When XPLINK applications are run under an LRR environment, the region size may need to be increased because an additional load module CELHV003 is kept in memory.

For ACTION=INIT, if the XPLINK= keyword is specified, valid values are YES or NO. If omitted, the default for the XPLINK= keyword is NO.

For ACTION=TERM, if the XPLINK= keyword is specified, it is ignored.

**Usage notes:**
1. The macro must be used in an assembler routine that is not Language Environment -conforming.
2. The contents of the following registers are destroyed by the macro invocation:
   - R14
   - R15: Upon return, contains the return code
   - R0
   - R1
3. The code generated by the macro expansion assumes that R13 has a standard RSA available.
4. One of the following return codes is put in R15 upon completion of the code generated by the CEELRR macro with ACTION=INIT:
Assembler considerations

0
Library routine retention was successfully initialized.

4
Library routine retention is already initialized. No action was taken.

8
Library routine retention was not initialized; the parameter list is not recognized.

12
Library routine retention was not initialized due to one of the following problems:
  • There was insufficient storage.
  • There was an error in an attempt to load CEEBINIT or CEEBLIBM.

16
Library routine retention was not initialized because Language Environment is currently initialized. This return code can occur in the following example scenarios:
  • A program that is running with Language Environment calls an assembler program that uses CEELRR with ACTION=INIT.
  • An assembler program calls IGZERRE to initialize a reusable environment, and then it uses CEELRR with ACTION=INIT.
  • A reusable environment is established with the RTEREUS runtime option and a call is made to an assembler program that uses CEELRR with ACTION=INIT.

20
Library routine retention was not initialized because the Language Environment preinitialized environment has been established and is dormant. This return code can occur in the following example scenarios:
  • An assembler program calls CEEPIPI to preinitialize Language Environment, and then it uses CEELRR with ACTION=INIT.
  • An assembler program uses the PL/I preinitialize program interface, and then it uses CEELRR with ACTION=INIT.

5. One of the following return codes is put in R15 upon completion of the code generated by the CEELRR macro with ACTION=TERM:

0
Library routine retention was successfully terminated. All resources associated with library routine retention were freed.

4
Library routine retention is not initialized. No action was taken.

8
Library routine retention was not terminated; the parameter list is not recognized.

16
Library routine retention was not terminated because Language Environment is currently initialized. This return code can occur in the following example scenarios:
  • A program that is running with Language Environment calls an assembler program that uses CEELRR with ACTION=TERM.
  • An assembler program calls IGZERRE with the initialize function, and then it uses CEELRR with ACTION=TERM.
  • A reusable environment is established with the RTEREUS runtime option and a call is made to an assembler program that uses CEELRR with ACTION=TERM.

20
Library routine retention was not terminated because the Language Environment preinitialized environment has been established and is dormant. This return code can occur in the following example scenarios:
• An assembler program calls CEEPIPI to preinitialize Language Environment, and then it uses CEELRR with ACTION=TERM.
• An assembler program uses the PL/I preinitialize program interface, and then it uses CEELRR with ACTION=TERM.

### Assembler macros

Language Environment provides the following macros to assist in the entry and exit of assembler routines, to map the CAA and DSA, to generate the appropriate fields in the program prolog area (PPA), to create assembler DLLs, and to use DLLs from assembler routines:

- **CEEENTRY** generates a Language Environment-conforming prolog. You must use CEEENTRY in conjunction with the following macros, except for CEELOAD. (See “CEEENTRY macro—Generate a Language-Environment-conforming prolog” on page 400 for syntax.)
- **CEETERM** generates a Language Environment-conforming epilog and terminates the assembler routine. (See “CEETERM macro — Terminate a Language Environment-conforming routine” on page 403 for syntax.)
- **CEECAA** generates a CAA mapping. (See “CEECAA macro — Generate a CAA mapping” on page 404 for syntax.)
- **CEEDSA** generates a DSA mapping. (See “CEEDSA macro — Generate a DSA mapping” on page 404 for syntax.)
- **CEEPPA** generates the appropriate fields in the PPA in your assembler routine. The fields describe the entry point of a Language Environment block. (See “CEEPPA macro — Generate a PPA” on page 404 for syntax.)
- **CEELOAD** loads a Language Environment-conforming assembler routine; the target of CEELOAD must be a subroutine. (See “CEELOAD macro — Dynamically load a Language Environment-conforming routine” on page 407 for syntax.)
- **CEEFETCH** dynamically loads a routine and returns information about a routine. (See “CEEFETCH macro — Dynamically load a routine” on page 409 for syntax.)
- **CEEFTCH** generates a FTCHINFO mapping. (See “CEEFTCH macro — Generate a FTCHINFO mapping” on page 413 for syntax.)
- **CEEGLOB** is used to extract the Language Environment product information at assembly-time. (See “CEEGLOB macro — Extract Language Environment product information” on page 415 for syntax.)
- **CEERELES** dynamically deletes a routine. (See “CEERELES macro — Dynamically delete a routine” on page 416 for syntax.)
- **CEEPSCALL** calls a Language Environment-conforming routine. It is similar to the CALL macro, except that it supports dynamic calls to routines in a DLL. (See “CEEPSCALL macro — Pass control to control sections at specified entry points” on page 417 for syntax.)
- **CEEPDDA** defines a data item in WSA, or declares a reference to an imported data item. (See “CEEPDDA macro — Define a data item in the writeable static area (WSA)” on page 419 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 “CEEPLDA macro — Returns the address of a data item defined by CEEPDDA” on page 420 for syntax.)

For a description of Assembler macros to assist in writing XPLINK assembler routines, see z/OS XL C/C++ Programming Guide.

**Note:** All keyword parameter values, such as YES, NO, ANY, must be specified in upper case. For example, MAIN=YES, AMODE=ANY.
CEEENTRY macro—Generate a Language-Environment-conforming prolog

CEEENTRY provides a Language Environment-conforming prolog. Code is generated in cooperation with the CEEPPA macro. (See “CEEPPA macro—Generate a PPA” on page 404 for syntax.) The macro generates reentrant code.

You must use CEEENTRY in conjunction with the macros CEETERM, CEECAA, CEEDSA, and CEEPPA. CEEENTRY assumes that the registers contain what is described in “Register conventions” on page 393 for assembler main routines.

To call an assembler routine from an existing Fortran program, or to make a static call from OS/VS COBOL or VS COBOL II, CEEENTRY must specify NAB=NO and MAIN=NO.

(name) CEEENTRY

PPA= label, AUTO= value

NAB= YES NO

MAIN= YES NO

ENCLAVE= YES NO

EXECOPS= YES NO

PARMREG= register

EXPORT= YES NO

BASE= register (register)

PLIST= HOST MVS TSO CICS IMS OS

RMODE= ANY 24 31

AMODE= ANY 24 31 ANY31

STKPROT= YES NO

name

The entry name (and the CSECT name, if this is the first call to CEEENTRY).
**PPA=**
The label of the corresponding PPA (Program Prolog Area) generated using the CEEPPA macro. If unspecified, the name “PPA” is used.

**AUTO=**
The total number of bytes (rounded up to a doubleword) used by prolog code for the DSA and local automatic variables that are to be allocated for the duration of this routine. If unspecified, the default is only the size of the DSA without any automatic variables. This default size is indicated by the label CEEDSASZ (the DSA by the CEEDSA macro). See “CEEDSA macro — Generate a DSA mapping” on page 404 for syntax.

**NAB=**

**YES**
Indicates that the previous save area has a NAB (next available byte) value. In general,

- If your routine is always called by a Language Environment-conforming assembler routine, specify NAB=YES.
- If your routine can be called by a non-Language Environment-conforming assembler routine, specify NAB=NO.

YES is the default.

**NO**
Indicates that the previous save area may not contain the NAB. Code to find the NAB is generated. This parameter is ignored if MAIN=YES is specified. You must specify MAIN=NO and NAB=NO to call an assembler routine from an existing Fortran application, or to make a static call from OS/VS COBOL or VS COBOL II.

**MAIN=**

**YES**
Indicates that the Language Environment environment should be brought up. Designates this assembler routine as the main routine in the enclave. YES is the default. If you specify MAIN=YES, you cannot specify register 2 as the base register for the module. MAIN=YES is not supported under CICS on releases earlier than CICS TS 3.1.

The following is accomplished by the macro invocation:

- The caller's registers (14 through 12) are saved in a DSA provided by the caller.
- The base register is set (see BASE= for more information).
- Register 12 is set with an address of CEECAA.
- Register 13 is set with an address of CEEDSA.
- PARMREG (Register 1 is the default) is set based on PLIST.
- All other registers are undefined.

YES is the default.

**NO**
Designates this assembler routine as a subroutine in the enclave. NO should be specified when the Language Environment environment is already active and only prolog code is needed. You must specify NAB=NO in order to call an assembler routine from an existing Fortran application, or to make a static call from OS/VS COBOL or VS COBOL II.

The following is accomplished by the macro invocation:

- The caller's registers (14 through 12) are saved in a DSA provided by the caller.
- The base register is set (see BASE= for more information).
- Register 13 is set with an address of CEEDSA.
- PARMREG is set (see PARMREG.)
- All other registers are undefined.
ENCLAVE=

YES
Indicates that Language Environment should always create a nested enclave for this program. ENCLAVE=YES can only be specified when MAIN=YES. The use of ENCLAVE=YES will result in increased storage and CPU usage. Most applications will not need a new enclave; therefore ENCLAVE=NO should be used.

NO
Indicates that a new enclave is not needed for this program. NO is the default.

EXECOPS=

YES
Indicates that the main routines are to honor runtime options on the inbound parameter string. This option is applicable only when MAIN=YES is in effect for the routine. The EXECOPS setting is ignored if MAIN=NO is specified. YES is the default.

NO
Indicates that there are no runtime options in the inbound parameter string. Language Environment considers the entire inbound parameter string as program arguments, but does not attempt to process runtime options and remove them from the inbound parameter string.

PARMREG=

Specifies the register to hold the inbound parameters. If you do not specify a value, register 1 is assumed.

For MAIN=YES, the value in the PARMREG is determined by PLIST. For MAIN=NO and PARMREG=1 (PARMREG defaults to 1), register 1 is restored from the save area that is passed to the routine. When MAIN=NO and PARMREG is not equal to 1, register 1 is used to load the specified PARMREG. Then, register 1 is used for the CEEENTRY expansion.

EXPORT=

Indicates whether this entry point will be exported.

NO
This entry point can only be called from other routines that are link-edited into the same program object. NO is the default.

YES
This entry point is marked as an exported DLL function. If you specify EXPORT=YES, then you must use the GOFF Assembler option.

If you want the exported name to be a long name or mixed case, follow the CEEENTRY macro with an ALIAS statement. For more details about DLLs, including full sample assembler DLL routines, see Chapter 4, "Building and using dynamic link libraries (DLLs)," on page 37.

For the entry 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.

BASE=

Establishes the registers that you specify here as the base registers for this module. If you do not specify a value, register 11 is assumed; register 12 cannot be used. When more than one register is specified, the registers must be separated by commas and enclosed in parentheses. The same register cannot be specified more than once.

PLIST=

Indicates that the main routines are to honor PLIST format on the inbound parameter string. This option is applicable only when MAIN=YES is in effect for the routine. The PLIST settings are ignored if MAIN=NO is specified.

The HOST format sets the specified PARMREG based on the environment in which the program is executing. For example, in an environment that assumes CEEENTRY defaults, register 1 is set equal to the address of a one word PLIST that contains the address of a field with a halfword-prefixed string of user parameters. To obtain the inbound parameter list as specified, use PLIST (OS).
HOST is the default.

**RMODE=**

Allows the specification of the modules CSECT RMODE setting. Valid settings for this option are ANY, 24 and 31. ANY is the default.

**AMODE=**

Allows the specification of the modules CSECT AMODE setting. Valid settings for this option are 24, 31, ANY31, and ANY. ANY is the default.

**STKPROT=**

Indicates whether this procedure has STACKPROTECT enabled.

**Usage notes:**

1. The CEEENTRY macro automatically sets the module to AMODE ANY and RMODE ANY. Therefore, when converting to Language Environment-conforming assembler, if data management macros had been coded using 24-bit addressing mode, they should be changed to use 31-bit addressing mode (see z/OS DFSMS Macro Instructions for Data Sets). If it is not possible to change all the modules making up the program to use 31-bit addressing mode (and none of the modules are already set to RMODE 24), then it will be necessary to use the RMODE=24 CEEENTRY option. Alternatively, the module can be set to RMODE 24 during the link-edit process. This is done by specifying the link-edit RMODE option on the invocation PARM or the SETOPT control statement.

2. Unless otherwise indicated, no register values should be expected to remain unchanged after the code generated by CEEENTRY has executed.

3. When more than one CEEENTRY 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 CEEENTRY macro.

**CEETERM macro — Terminate a Language Environment-conforming routine**

CEETERM provides a Language Environment-conforming epilog and is used to terminate, or return from, a Language Environment-conforming routine. If used with a main entry, the appropriate call is made to Language Environment termination routines.

```plaintext
CEETERM

name
RC= return_code
MODIFIER= modifier
MF=L
(ctrl_addr)
```

**name**

The entry name (and the CSECT name, if this is for a main entry).

**RC=**

The *return code* that is to be placed into R15 after the MODIFIER is added to it, if terminating a main routine. If returning from a Language Environment subroutine, the return code itself is placed into R15, without MODIFIER being added to it. *return code* can be a fixed constant, variable, or register 2–12.

**MODIFIER=**

The return code *modifier* that is multiplied by the appropriate value (based upon the operating system), added to the return code, and placed into R15 (if terminating a main routine). The MODIFIER is independently placed into R0. Modifier can be a fixed constant, variable, or register 2-12.

**MF=L**

Indicates the list form of the macro. A remote control program parameter list for the macro is defined, but the service is not invoked. The list form of the macro is usually used in conjunction with the execute form of macro.
MF=(E, ctrl_addr)
Indicates the execute form of the macro. The service is invoked using the remote control program parameter list addressed by ctrl_addr (normally defined by the list form of the macro, it cannot be register 0).

Usage notes:
1. The MF=L and the MF=(E, ctrl_addr) parameters cannot both be coded for the same macro invocation. If neither is coded, the immediate form of the macro is used. The immediate form generates an inline parameter list, and generates nonreentrant code.
2. The address of the name can be specified as a register using parentheses ()
3. The macro invocation destroys the registers R1, R14, and R15.
4. MF=L and MF=(E, ctrl_addr) can only be used when CEEENTRY MAIN=YES has been specified. These parameters are not necessary when CEEENTRY MAIN=NO has been specified; in that environment, CEETERM automatically generates reentrant code.

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

CEEDSA is used to generate a dynamic save area (DSA) mapping. This macro has no parameters, and no label can be specified. The minimum size of the DSA is contained in an assembler EQUATE CEEDSASZ. CEEDSA is required for the CEEENTRY macro.

CEEPPA macro — Generate a PPA

CEEPPA is used to generate the Language Environment program prolog area (PPA). The PPA defines constants that describe the entry point of a Language Environment block. It is generated at the time of assembly; one PPA is generated per entry point. The CEEPPA macro is required for the CEEENTRY macro.
**label**

The name of the PPA. If you specified a name for PPA in the CEEENTRY macro, you must specify the same name here. If you did not specify a name for PPA in the CEEENTRY macro, you must specify PPA (the CEEENTRY default PPA label) as the name here.

**LIBRARY=**

Indicates whether the routine is a Language Environment library routine. Valid values for LIBRARY are YES and NO. If you do not specify a value, NO is used. Use of this IBM-supplied default is recommended.

**PPA2=**

Instructs the macro to generate a PPA2 or suppress the generation of the PPA2. A PPA2 is a program prolog area that defines constants for the CSECT. Only one is used, independent of the number of entry points. Valid values for PPA2 are YES and NO.

The default is YES, which generates a PPA2 field.

**EXTPROC=**

Indicates if this routine is an external procedure or an internal procedure. Valid values for EXTPROC are YES and NO. The default is YES, which indicates that the block is an external procedure.
Assembler considerations

TSTAMP=
Indicates whether a time stamp, indicating the date and time of assembly, should be generated. Valid values for TSTAMP are YES and NO. The default is YES is used and a time stamp is generated.

PEP=
Indicates whether this entry point is primary or secondary. A secondary entry point is an alternative entry point. Some Language Environment facilities, such as CEE3DMP, report information that is based on the primary entry point only.

Valid values for PEP are YES and NO. The default is YES, which indicates that this is a primary entry point (PEP).

INSTOP=
Indicates whether time spent in this routine should be attributed to the program (rather than to the system). Valid values for INSTOP are YES and NO. The default is NO, which indicates that time should be attributed to the system. The information is intended to be used by application performance analysis tools.

EXITDSA=
Indicates whether the code should gain control on GOTO out of block. Valid values for EXITDSA are YES and NO. The default is NO, which indicates that the code does not gain control for GOTO out of block. Use of this IBM-supplied default is recommended.

OWNEXM=
Specifies whether this routine should participate in condition handling according to the exception model of its own member language (OWNEXM=YES), or according to the exception model inherited from the caller's member language (OWNEXM=NO). Valid values for OWNEXM are YES and NO. The default is YES. Use of this IBM-supplied default is recommended.

EPNAME=
Indicates the entry point name. The default is the name of the CSECT is used.

VER=
The version number for the routine. This field is not interrogated by Language Environment. Valid values for VER are 1 through 99. The default is 1.

REL=
The release number for the routine. This field is not interrogated by Language Environment. Valid values for REL are 1 through 99. If you do not specify a value, 1 is used.

MOD=
The modification level for the routine. This field is not interrogated by Language Environment. Valid values for MOD are 1 through 99. The default is 1.

DSA=YES
Indicates whether this procedure has a DSA. Valid values for DSA are YES and NO. The default is YES, which indicates that the code has an associated DSA. Use of this IBM-supplied default is recommended.

SERVICE=
Indicates the service level string for the routine. The service string length and contents are located following the time stamp and version information. The first 7 bytes of the service level string is treated as character data for the Service column of a traceback. When the SERVICE keyword is in use, the time stamp is generated automatically, the TSTAMP option is defaulted to YES even when the user specified TSTAMP=NO. The SERVICE keyword can only be specified on the first CEEPPA macro in the assembler source. All other instances of the keyword are ignored.

VRSMASK=
The Vector Registers save bit mask field in hexadecimal format. Valid values for VRSMASK are 00 through FF. VRSMASK and VRSLOCR must be provided at the same time for the VRs optional area.

VRSLOCR=
The Vector Registers locator field in hexadecimal format. Valid values for VRSLOCR are 00 through FF. VRSMASK and VRSLOCR must be provided at the same time for the VRs optional area.

STKPROT=
Indicates whether this procedure has STACKPROTECT enabled.
CEELOAD macro — Dynamically load a Language Environment-conforming routine

CEELOAD is used to dynamically load a Language Environment-conforming routine. It does not create a nested enclave, so the target of CEELOAD must be a subroutine.

There is no corresponding service to delete Language Environment-conforming routines. You should not use system services to delete modules that you load using CEELOAD; during thread (if SCOPE=THREAD) or enclave (if SCOPE=ENCLAVE) termination, Language Environment deletes modules loaded by CEELOAD.

Using CEELOAD imposes restrictions on further dynamic loading or dynamic calls or fetches; results are unpredictable if these rules are violated.

- You cannot dynamically load a routine with CEELOAD that has already been dynamically loaded by CEELOAD or has been fetched or dynamically called.
- You cannot fetch or dynamically call a routine that has already been dynamically loaded by CEELOAD.

If CEELOAD completes successfully, the address of the loaded routine is found in R15. You can then invoke the routine using BALR 14,15 (or BASSM 14,15).

Language Environment returns the address of the target routine with the high-order bit indicating the addressing mode (AMODE) of the routine. Language Environment-enabled programs return in the AMODE in which they are entered. Because Language Environment does not provide any AMODE switching on behalf of the target routine, you must provide any necessary AMODE switching code.

The macro invocation destroys the following registers:

- R0
- R1
- R14
- R15 (upon return, contains the target address)

When the macro code is expanded and run, the following assumptions are made:

- R12 points to the CAA.
- R13 has a standard Language Environment DSA available.

**label**

The assembler label you give to this invocation of the macro. A label is required if MF=L; otherwise it is optional.

**NAME=**

The name of the entry point to be loaded by Language Environment. If MF=I or MF=L, you must specify either NAME or NAMEADDR, but not both.

**NAMEADDR=**

The address of a halfword-prefixed name that should be loaded by Language Environment. This can be an A-type address or a register (register 2 through 11). If MF=I or MF=L, you must specify either NAME or NAMEADDR, but not both. The address of the name can be specified as a register using parentheses ().
**SCOPE=THREAD**
Indicates that the load is to be scoped to the thread level. Modules loaded at the thread level are deleted automatically at thread termination.

**SCOPE=ENCLAVE**
Indicates that the load is to be scoped to the enclave level. Modules loaded at the enclave level are deleted automatically at enclave termination. **SCOPE=ENCLAVE** is the default.

**FEEDBACK=**
The name of a variable to contain the resulting 12-byte feedback token. If you omit this parameter, any nonzero feedback token that results is signaled. The following symbolic conditions might be returned from this service:

<table>
<thead>
<tr>
<th>Symbolic feedback code</th>
<th>Severity</th>
<th>Message number</th>
<th>Message text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE000</td>
<td>0</td>
<td>—</td>
<td>The service completed successfully.</td>
</tr>
<tr>
<td>CEE3DC</td>
<td>3</td>
<td>3500</td>
<td>Not enough storage was available to load module-name.</td>
</tr>
<tr>
<td>CEE3DD</td>
<td>3</td>
<td>3501</td>
<td>The module module-name was not found.</td>
</tr>
<tr>
<td>CEE3DE</td>
<td>3</td>
<td>3502</td>
<td>The module name module-name was too long.</td>
</tr>
<tr>
<td>CEE3DF</td>
<td>3</td>
<td>3503</td>
<td>The load request for module module-name was unsuccessful.</td>
</tr>
<tr>
<td>CEE39K</td>
<td>1</td>
<td>3380</td>
<td>The target load module was not recognized by Language Environment.</td>
</tr>
<tr>
<td>CEE38M</td>
<td>3</td>
<td>3350</td>
<td>CEE3ADM or CEE3MBR could not find the event handler.</td>
</tr>
<tr>
<td>CEE38N</td>
<td>3</td>
<td>3351</td>
<td>CEE3ADM or CEE3MBR could not properly initialize the event handler.</td>
</tr>
<tr>
<td>CEE38V</td>
<td>2</td>
<td>3359</td>
<td>The module or language list is not supported in this environment.</td>
</tr>
</tbody>
</table>

**MF=I**
Indicates the immediate form of the macro. The immediate form generates an inline parameter list, and generates nonreentrant code.

**MF=L**
Indicates the list form of the macro. A remote control program parameter list for the macro is defined, but the service is not invoked. The list form of the macro is usually used in conjunction with the execute form of the macro.

**MF=(E, ctrl_addr)**
Indicates the execute form of the macro. The service is invoked using the remote control program parameter list addressed by ctrl_addr (normally defined by the list form of the macro).

Only one of the MF=I, MF=L, or MF=(E, ctrl_addr) parameters can be coded for the same macro invocation. If none is coded, the immediate form of the macro is used.

The following example illustrates an invocation of the CEELOAD macro.

```assembly
CLOADTST CEEENTRY MAIN=YES,PPA=LEPPA,AUTO=DSALGTH
********************************************************
* Copy parameters to be passed to CEELOAD
********************************************************
MVC LOADPL(PLLEN),PLLIST
********************************************************
* Invoke CEELOAD to load module HIWORLD
********************************************************
CEELOAD MF=(E,LOADPL) LOAD ROUTINE
********************************************************
* Pass control to HIWORLD
********************************************************
BALR 14,15 INVOKE ROUTINE
```
**Usage notes:**

1. Language Environment issues the appropriate load command according to the Language Environment search order (described in “Program library definition and search order” on page 69) and performs the necessary dynamic updates to accommodate the new load module.

2. Language Environment performs any language-related initialization required.

3. You cannot use CEELOAD to load a program object which was created using the program management binder. You can, however, use CEEFETCH for loading program objects.

4. You cannot use CEELOAD to load C++ modules, because C++ modules are always compiled RENT and have writable static that is not switched when control passes between functions.

5. `#pragma linkage (xxx,fetchable)` should not be used. If a module is linked with `#pragma linkage (xxx,fetchable)`, it will have CEESTART as an entry point, which is not allowed, and the module could have writable static requirements that would not be handled using CEELOAD.

   When using CEELOAD to load a C module, the function or functions within this module must not be designated as fetchable. The `#pragma linkage (xxx,fetchable)` directive should not be coded in the module. Instead, such modules should be fetched using the `fetch()` function.

6. `#pragma linkage (xxx,COBOL)` should not be used.

7. For C users, the load module entry point must be the function name, and cannot be CEESTART (nested environment initialization causes Language Environment to abend). You cannot use CEELOAD to load any function that uses writable static. The module must be built NORENT and the entry point must be a C function, not CEESTART.

8. This macro should not be used for DLLs.

### CEEFETCH macro — Dynamically load a routine

CEEFETCH is used to dynamically load a routine.

Use the CEERELES macro to delete routines that are loaded with CEEFETCH. Do not use system services to delete modules that you load using CEEFETCH; during thread (if SCOPE=THREAD), enclave (if SCOPE=ENCLAVE), or process (if SCOPE=PROCESS) termination, Language Environment deletes modules that are loaded by CEEFETCH.

If CEEFETCH completes successfully, the address of the target routine is found in R15. You can then invoke the routine using the BALR 14,15 (or BASSM 14,15) instruction.

Language Environment returns the address of the target routine with the high-order bit indicating the addressing mode (AMODE) of the routine. Language Environment-enabled programs return in the AMODE in which they are entered. Because Language Environment does not provide any AMODE switching on behalf of the target routine, you must provide any necessary AMODE switching code.
Assembler considerations

For example:

```
LA     2,RESET  SAVE BRANCH ADDRESS AND CURRENT
BSM    2,0      AMODE IN REGISTER 2
BASSM 14,15     CALL COBOL PROGRAM
BSM    0,2      BRANCH AND RESTORE AMODE FROM REG. 2
RESET DS     OH
```

The macro invocation destroys the following registers:

- R0
- R1
- R14
- R15 (upon return, contains the target address)

When the macro code is expanded and run, the following assumptions are made:

- R12 points to the CAA.
- R13 has a standard Language Environment DSA available.

---

**label**

The assembler label that you give to this invocation of the macro. A label is required if MF=L is specified; otherwise, it is optional.

**NAME=name**

The name of the entry point to be loaded by Language Environment. The maximum length of `name` is eight characters. If a longer name is needed, the NAMEADDR parameter must be used. You cannot specify NAME and NAMEADDR together.

**NAMEADDR=nameaddr**

The address of a halfword-prefixed name that should be loaded by Language Environment. A halfword

prefix name is a string where the first two bytes identify the length of a name string and are followed by the name string itself. This can be an A-type address or a register (register 2 through 11). The
address of the name can be specified as a register using parentheses (). The maximum length of the name is 1023 characters. You cannot specify NAME and NAMEADDR together.

**ENTRYPT=entrypt**
The name of a fullword address variable that contains the entry point for a module that was previously loaded or the register (enclosed in parentheses) containing the entry point for a module that was previously loaded. The NAME and NAMEADDR keywords are mutually exclusive with ENTRYPT. The SEARCH keyword is ignored when ENTRYPT is specified. The FTCINFO keyword is required when ENTRYPT is specified. A corresponding “delete” using CEERELES can be done if CEEFETCH returns successfully.

**SCOPE=THREAD**
Indicates that the load is to be scoped to the thread level. Modules that are loaded at the thread level are deleted automatically at thread termination.

**SCOPE=ENCLAVE**
Indicates that the load is to be scoped to the enclave level. Modules that are loaded at the enclave level are deleted automatically at enclave termination. It is the default.

**SCOPE=PROCESS**
Indicates that the load is to be scoped to the process level. Modules that are loaded at the process level are deleted automatically at process termination.

**TOKEN=token**
The name of a variable to contain the resulting 4-byte token. This variable must be passed to the CEERELES macro if the load module is to be deleted. If MF=I or MF=L are specified, you must specify TOKEN.

**SEARCH=MVS**
Indicates that only the MVS file system is to be searched for the load module. It is the default.

**SEARCH=HFS**
Indicates that only the HFS file system is to be searched for the load module.

**SEARCH=MVSHFS**
Indicates that the MVS file system is to be searched first and then the HFS file system for the load module.

**SEARCH=HFSMVS**
Indicates that the HFS file system is to be searched first and then the z/OS file system for the load module.

**FEEDBACK=fbcode**
The name of a variable to contain the resulting 12-byte feedback token. If you omit this parameter, any nonzero feedback token that results is signaled. The following symbolic conditions might be returned from this service:

<table>
<thead>
<tr>
<th>Symbolic feedback code</th>
<th>Severity</th>
<th>Message number</th>
<th>Message text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE000</td>
<td>0</td>
<td></td>
<td>The service completed successfully.</td>
</tr>
<tr>
<td>CEE3DC</td>
<td>3</td>
<td>3500</td>
<td>Not enough storage was available to load module-name.</td>
</tr>
<tr>
<td>CEE3DD</td>
<td>3</td>
<td>3501</td>
<td>The module module-name was not found.</td>
</tr>
<tr>
<td>CEE3DE</td>
<td>3</td>
<td>3502</td>
<td>The module name module-name was too long.</td>
</tr>
<tr>
<td>CEE3DF</td>
<td>3</td>
<td>3503</td>
<td>The load request for module module-name was unsuccessful.</td>
</tr>
<tr>
<td>CEE39K</td>
<td>1</td>
<td>3380</td>
<td>The target load module was not recognized by Language Environment.</td>
</tr>
<tr>
<td>CEE38M</td>
<td>3</td>
<td>3350</td>
<td>CEE3ADM or CEE3MBR could not find the event handler.</td>
</tr>
</tbody>
</table>
### Symbolic Feedback Code

<table>
<thead>
<tr>
<th>Symbolic Feedback Code</th>
<th>Severity</th>
<th>Message Number</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE38N</td>
<td>3</td>
<td>3351</td>
<td>CEE3ADM or CEE3MBR could not properly initialize the event handler.</td>
</tr>
<tr>
<td>CEE3N9</td>
<td>2</td>
<td>3817</td>
<td>The member event handler did not return a usable function pointer.</td>
</tr>
<tr>
<td>CEE38V</td>
<td>2</td>
<td>3359</td>
<td>The module or language list is not supported in this environment.</td>
</tr>
<tr>
<td>CEE3DV</td>
<td>3</td>
<td>3519</td>
<td>The version that is specified in the CEEFTCH control block passed to the CEEFETCH macro is not supported.</td>
</tr>
<tr>
<td>CEE3QS</td>
<td>1</td>
<td>3932</td>
<td>The system service CSVQUERY failed with return code &lt;return_code&gt; and reason code 0.</td>
</tr>
</tbody>
</table>

### MF=I
- Indicates the immediate form of the macro. The immediate form generates an inline parameter list and generates nonreentrant code. This is the default value.

### MF=L
- Indicates the list form of the macro. A remote control program parameter list for the macro is defined, but the service is not invoked. The list form of the macro is usually used with the execute form of the macro.

### MF=(E, ctrl_addr)
- Indicates the execute form of the macro. The service is invoked using the remote control program parameter list addressed by ctrl_addr (usually defined by the list form of the macro).

Only one of the MF=I, MF=L, or MF=(E, ctrl_addr) parameters can be coded for the same macro invocation. If none is coded, the immediate form of the macro is used.

### FTCHINFO=ftchinfo
- The name of a fullword address variable that contains the address of a previously allocated FTCHINFO storage area or the register (enclosed in parentheses) containing the address of a preallocated FTCHINFO storage area, where the resulting module information is to be stored. The user must set the CEEFTCH_VERSION field in the FTCHINFO storage area. This keyword is useful for retrieving information about a target module whose characteristics are unknown. If the module is identified as a Language Environment conforming AMODE 24 or AMODE 31 subroutine, then processing would be as normal, otherwise only a load of the target is attempted. Only an AMODE 24 or AMODE 31 target module can be recognized as a DLL. An AMODE 24 or AMODE 31 COBOL target is classified as a subroutine and follows the normal processing. The CEEFETCH macro provides the mapping for the FTCHINFO storage area. See the “CEEFETCH macro — Generate a FTCHINFO mapping” on page 413 description for details about its contents.

### Usage Notes:

1. Language Environment issues the appropriate load command according to Language Environment search order (described in “Program library definition and search order” on page 69), and performs the necessary dynamic updates to accommodate the load module.
2. Language Environment performs any language-related initialization required.
3. Any COBOL, PL/I, or Fortran module that is fetched, dynamically called, or CEEFETCHed more than once must be reentrant.
4. When using CEEFETCH to fetch a C module, the C module must contain `#pragma linkage (xxx, fetchable)`. For exceptions to this rule, see the `fetch()` information in z/OS XL C/C++ Runtime Library Reference.
5. CEEFETCH can be used in a non-XPLINK Assembler program to fetch an XPLINK-compiled module. The fetched XPLINK-compiled module must contain `#pragma linkage(xxx, fetchable)`. The address of the target routine that is returned by CEEFETCH in R15 contains any necessary glue code.
to call an XPLINK routine from non-XPLINK, and can still be invoked using BALR 14,15. All rules and
restrictions on the environment that is imposed by XPLINK still apply. See Chapter 3, “Using Extra
Performance Linkage (XPLINK),” on page 25.

6. Do not use CEEFETCH for DLLs.

7. If CEEFETCH is used to fetch a C++ module, the C++ module must contain #pragma
   linkage(xxx,fetchable) directive and must be declared extern "C". See z/OS XL C/C++ Runtime Library
   Reference for more information.

8. The Fortran compilers do not conform to the Language Environment interface conventions, thus
   Fortran targets will not be recognized as Language Environment conforming. For more information,
   see the Fortran migration guide.

9. Do not reuse the contents that are returned in R15 after a call to CEEFETCH as input to a follow-up
call to CEEFETCH. This scenario could result in the issue of the CEE3932W message.

10. In a multithread environment, using a CEEFETCH / BALR (or BASSM) / CEERELES sequence on more
    than one thread is not supported if the target routine is the same COBOL routine, even when the
    COBOL routine is enabled for multithreading. COBOL does not allow a CANCEL of a routine that is
    active on another thread.

11. In a multithread environment, using a CEEFETCH / BALR (or BASSM) / CEERELES sequence on more
    than one thread is not supported if the target routine is the same PL/I routine.

12. The SCOPE of the CEEFETCH is with respect to the load module only. Other storage and data that is
    associated with the module, such as the function descriptor and writeable static area (WSA), are
    scoped to the enclave level. (If C/C++, PL/I, and certain features of other compilers are used, function
    descriptors and writeable static areas might exist.)

CEEFETCH macro — Generate a FTCHINFO mapping

CEEFETCH is used to generate a mapping for the FTCHINFO storage area. Module information can be
returned from the CEEFETCH macro in a FTCHINFO storage area. No label can be specified for this macro.

The FLAG information provided for AMODE 64 modules is limited to the AMODE, if Language
Environment-conforming, if XPLINK, and if SEGMENTED. A target that is recognized as an AMODE 24 or
AMODE 31 DLL, will have the MAIN and SUB bits turned OFF. The EP address for an AMODE 31 target will
have the high order bit turned off and the EP64 address for an AMODE 64 target will have the low-order
bit turned off.

When the ENTRYPT keyword is used with CEEFETCH in the CICS environment (running on the QR TCB),
the load point address, load length, and module segment information will not be provided in the
FTCHINFO storage area. The module segment information will also not be provided when the FTCHINFO
keyword is used without ENTRYPT keyword in the CICS environment.

The user must set the CEEFETCH_VERSION field in CEEFETCH before invoking the CEEFETCH macro to
return module information. (Currently the only valid value for CEEFETCH_VERSION is 1.) An unsupported
version will result in the CEE3DV feedback code from CEEFETCH. See the following CEEFETCH tables for
supported mapping versions.

---

DSECT=YES
Indicates that a DSECT mapping should be generated. This is the default for the mapping if the DSECT
option is not specified.

DSECT=NO
Indicates that a data area mapping should be generated.

The following tables show the format of the CEEFETCH mapping Version 1 (CEEFETCH_VERSION = 1).
### Table 64: Structure of version 1 CEEFTCH

<table>
<thead>
<tr>
<th>OFFSET Dec</th>
<th>OFFSET Hex</th>
<th>Type</th>
<th>Len</th>
<th>Name (Dim)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0)</td>
<td>STRUCTURE</td>
<td>64</td>
<td>CEEFTCH</td>
<td>Start of CEEFTCH</td>
</tr>
<tr>
<td>0</td>
<td>(0)</td>
<td>CHARACTER</td>
<td>8</td>
<td>CEEFTCH_EYE_CATCHER</td>
<td>eyecatcher</td>
</tr>
<tr>
<td>8</td>
<td>(8)</td>
<td>UNSIGNED</td>
<td>2</td>
<td>CEEFTCH_VERSION</td>
<td>Version requested</td>
</tr>
<tr>
<td>10</td>
<td>(A)</td>
<td>BIT(8)</td>
<td>1</td>
<td>CEEFTCH_FLAGS1</td>
<td>CEEFTCH flags1</td>
</tr>
<tr>
<td>10</td>
<td>(A)</td>
<td>BIT(1)</td>
<td>1</td>
<td>CEEFTCH_A24</td>
<td>X'80' target is AMODE 24</td>
</tr>
<tr>
<td>10</td>
<td>(A)</td>
<td>BIT(1) POS(2)</td>
<td>1</td>
<td>CEEFTCH_A31</td>
<td>X'40' target is AMODE 31</td>
</tr>
<tr>
<td>10</td>
<td>(A)</td>
<td>BIT(1) POS(3)</td>
<td>1</td>
<td>CEEFTCH_A64</td>
<td>X'20' target is AMODE 64</td>
</tr>
<tr>
<td>10</td>
<td>(A)</td>
<td>BIT(1) POS(4)</td>
<td>1</td>
<td>CEEFTCH_XPLINK</td>
<td>X'10' target is XPLINK</td>
</tr>
<tr>
<td>10</td>
<td>(A)</td>
<td>BIT(1) POS(5)</td>
<td>1</td>
<td>CEEFTCH_LE</td>
<td>X'08' target is Language Environment conforming</td>
</tr>
<tr>
<td>10</td>
<td>(A)</td>
<td>BIT(1) POS(6)</td>
<td>1</td>
<td>CEEFTCH_MAIN</td>
<td>X'04' target is MAIN</td>
</tr>
<tr>
<td>10</td>
<td>(A)</td>
<td>BIT(1) POS(7)</td>
<td>1</td>
<td>CEEFTCH_SUB</td>
<td>X'02' target is a SUB</td>
</tr>
<tr>
<td>10</td>
<td>(A)</td>
<td>BIT(1) POS(8)</td>
<td>1</td>
<td>CEEFTCH_DLL</td>
<td>X'01' target is DLL</td>
</tr>
<tr>
<td>11</td>
<td>(B)</td>
<td>BIT(8)</td>
<td>1</td>
<td>CEEFTCH_FLAGS2</td>
<td>CEEFTCH flags2</td>
</tr>
<tr>
<td>11</td>
<td>(B)</td>
<td>BIT(1)</td>
<td>1</td>
<td>CEEFTCH_SEGMENTED</td>
<td>X'80' target module is divided into multiple initial load segments (deferred load segments, if any, are not counted)</td>
</tr>
<tr>
<td>11</td>
<td>(B)</td>
<td>BIT(1) POS(2)</td>
<td>1</td>
<td>CEEFTCH_CICS</td>
<td>X'40' CICS environment</td>
</tr>
<tr>
<td>12</td>
<td>(C)</td>
<td>SIGNED</td>
<td>4</td>
<td>*</td>
<td>Available</td>
</tr>
<tr>
<td>16</td>
<td>(10)</td>
<td>ADDRESS</td>
<td>8</td>
<td>CEEFTCH_CEESTART64</td>
<td>Address of 64bit CEESTART</td>
</tr>
<tr>
<td>16</td>
<td>(10)</td>
<td>SIGNED</td>
<td>4</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>(14)</td>
<td>ADDRESS</td>
<td>4</td>
<td>CEEFTCH_CEESTART</td>
<td>Address of 31-bit CEESTART</td>
</tr>
<tr>
<td>24</td>
<td>(18)</td>
<td>ADDRESS</td>
<td>8</td>
<td>CEEFTCH_MOD64</td>
<td>Address of 64-bit target</td>
</tr>
<tr>
<td>24</td>
<td>(18)</td>
<td>SIGNED</td>
<td>4</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>(1C)</td>
<td>ADDRESS</td>
<td>4</td>
<td>CEEFTCH_MOD</td>
<td>Address of 31-bit target</td>
</tr>
<tr>
<td>32</td>
<td>(20)</td>
<td>SIGNED</td>
<td>8</td>
<td>CEEFTCH_MOD_LEN64</td>
<td>Length of 64-bit target</td>
</tr>
<tr>
<td>32</td>
<td>(20)</td>
<td>SIGNED</td>
<td>4</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>(24)</td>
<td>SIGNED</td>
<td>4</td>
<td>CEEFTCH_MOD_LEN</td>
<td>Length of 31-bit target</td>
</tr>
<tr>
<td>40</td>
<td>(28)</td>
<td>ADDRESS</td>
<td>8</td>
<td>CEEFTCH_EP64</td>
<td>Address of 64-bit EntryPt</td>
</tr>
<tr>
<td>40</td>
<td>(28)</td>
<td>SIGNED</td>
<td>4</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>(2C)</td>
<td>ADDRESS</td>
<td>4</td>
<td>CEEFTCH_EP</td>
<td>Address of 31-bit EntryPt</td>
</tr>
<tr>
<td>48</td>
<td>(30)</td>
<td>UNSIGNED</td>
<td>8</td>
<td>*</td>
<td>Available</td>
</tr>
<tr>
<td>56</td>
<td>(38)</td>
<td>UNSIGNED</td>
<td>8</td>
<td>*</td>
<td>Available</td>
</tr>
</tbody>
</table>

### Table 65: Cross reference for version 1 CEEFTCH

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEFTCH</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

414 z/OS: Language Environment Programming Guide
<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEFTCH_A24</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_A31</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_A64</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_CEESTART</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_CEESTART64</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>CEEFTCH_CICS</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_DLL</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_EP</td>
<td>2C</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_EP64</td>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>CEEFTCH_EYE_CATCHER</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>CEEFTCH_FLAGS1</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>CEEFTCH_FLAGS2</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>CEEFTCH_LE</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_MAIN</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_MOD</td>
<td>1C</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_MOD_LEN</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_MOD_LEN64</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>CEEFTCH_MOD64</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>CEEFTCH_SEGMENTED</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_SUB</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>CEEFTCH_VERSION</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>CEEFTCH_XPLINK</td>
<td>A</td>
<td>3</td>
</tr>
</tbody>
</table>

**CEEGLOB macro — Extract Language Environment product information**

CEEGLOB is used to generate global symbols that provide Language Environment product information at assembly time.

```
label

CEEGLOB

label
```

*label* is an optional assembler label that a user can give to this invocation of the macro.

The CEEGLOB macro generates the following global assembler variables to match the information returned by CEEGPID:

- **&CEEPRO (alias &GPRO)**
  Product number
- **&CEEVER (alias &GVER)**
  Product version
- **&CEGREL (alias &GREL)**
  Product release
- **&CEEGMOD (alias &GMOD)**
  Product modification level
&CEEGENV (alias &GENV)

OS environment from which the macro was invoked. Set to 3 for z/OS.

These global assembler variables can be tested and used at assembly time to verify availability of services and function that require specific levels of Language Environment or operating systems.

Note: If you need to make a decision concerning Language Environment and z/OS levels at runtime instead of assembly time, use CEEGPID instead. See z/OS Language Environment Programming Reference for a description of CEEGPID.

CEERELES macro — Dynamically delete a routine

CEERELES is used to dynamically delete a routine. The macro invocation destroys the following registers:

- R0
- R1
- R14
- R15

When the macro code is expanded and run, the following assumptions are made:
- R12 points to the CAA.
- R13 has a standard Language Environment DSA available.

```
label

CEERELES

TOKEN= token

FEEDBACK= fbcode

MF=I

MF=L

MF=(E, ctrl_addr)
```

**label**

The assembler label you give to this invocation of the macro. A label is required if MF=L is specified; otherwise, it is optional.

**TOKEN=token**

The name of a variable that contains the token returned by the CEEFETCH macro. If MF=I or MF=L, you must specify TOKEN.

**FEEDBACK=fbcode**

The name of a variable to contain the resulting 12-byte feedback token. If you omit this parameter, any nonzero feedback token that results is signaled. The following symbolic conditions might be returned from this service:

<table>
<thead>
<tr>
<th>Symbolic feedback code</th>
<th>Severity</th>
<th>Message number</th>
<th>Message text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE000</td>
<td>0</td>
<td>—</td>
<td>The service completed successfully.</td>
</tr>
<tr>
<td>CEE38N</td>
<td>3</td>
<td>3351</td>
<td>An event handler was unable to process a request successfully.</td>
</tr>
<tr>
<td>CEE39K</td>
<td>1</td>
<td>3380</td>
<td>The target load module was not recognized by Language Environment.</td>
</tr>
<tr>
<td>CEE3DG</td>
<td>3</td>
<td>3504</td>
<td>Delete service request for module-name was unsuccessful.</td>
</tr>
</tbody>
</table>
## Symbolic feedback code

<table>
<thead>
<tr>
<th>Symbolic feedback code</th>
<th>Severity</th>
<th>Message number</th>
<th>Message text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE3E0</td>
<td>3</td>
<td>3520</td>
<td>The token passed to the CEERELES macro was invalid.</td>
</tr>
</tbody>
</table>

**MF=I**
Indicates the immediate form of the macro. The immediate form generates an inline parameter list, and generates nonreentrant code. This is the default.

**MF=L**
Indicates the list form of the macro. A remote control program parameter list for the macro is defined, but the service is not invoked. The list form of the macro is usually used with the execute form of the macro.

**MF=(E, ctrl_addr)**
Indicates the execute form of the macro. The service is invoked using the remote control program parameter list addressed by `ctrl_addr` (usually defined by the list form of the macro).

Only one of the `MF=I`, `MF=L`, or `MF=(E, ctrl_addr)` parameters can be coded for the same macro invocation. If none is coded, the immediate form of the macro is used.

**Usage notes:**

1. Language Environment issues the appropriate operating system delete command and performs the necessary dynamic updates to accommodate the deleted load module.
2. Language Environment performs any language-related cleanup required.
3. This macro should not be used for DLLs.
4. In a multithread environment, using a CEEFETCH / BALR (or BASSM) / CEERELES sequence on more than one thread is not supported if the target routine is the same COBOL routine, even when the COBOL routine is enabled for multithreading. COBOL does not allow a CANCEL of a routine that is active on another thread.
5. In a multithread environment, using a CEEFETCH / BALR (or BASSM) / CEERELES sequence on more than one thread is not supported if the target routine is the same PL/I routine.

### CEEPCALL macro — Pass control to control sections at specified entry points

The CEEPCALL macro passes control to a control section at a specified entry point. The target of CEEPCALL can be resolved either statically (link-edited with the same program object) or dynamically (imported from a DLL). The only required positional parameter is the name of the called entry point. This name is case-sensitive, and can be up to 255 characters in length. The optional parameter list will be pointed to by General Purpose Register (GPR) 1.

Since only REENTRANT Assembler code is supported by this macro, it must be specified as a combination of LIST and EXECUTE forms so that the parameter list can be built in automatic (that is, stack) storage.

The CEEPCALL macro does not generate any return codes. A return code may be placed in GPR 15 by the called program.

GPRs 1, 14, and 15 are not preserved by this macro.
**label**  
Optional symbol 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 optional parameters to be passed to the called program, separated by commas. In the list form, these are specified as A-type addresses, and in the execute form are RX-type addresses or specified as registers (2) - (12).

To create the parameter list, the calling program creates a list of addresses of each parameter in the order designated. In the execute form of the macro, GPR 1 contains the address of the parameter list when the program receives control. (If no parameters are coded, GPR 1 is not altered.) See Figure 103 on page 419.

**VL**  
Code VL only if the called program can be passed a variable number of parameters. VL causes the high-order bit of the last address parameter to be set to 1; the bit can be checked to find the end of the list.

**MF=L**  
Creates the list form of the CEEPCALL macro to construct a nonexecutable problem program parameter list. This list form generates only ADCONs of the address parameters. You should refer to this problem program parameter list in the execute form of a CEEPCALL macro.

**MF=(E,prob_addr)**  
Creates the execute form of the CEEPCALL macro, which can refer to and modify a remote problem program parameter list. Only executable instructions and a function descriptor representing the entry point are generated.

**Usage notes:**

1. This macro requires the GOFF Assembler option
2. This macro requires the binder to link edit, and the RENT and DYNAM(DLL) binder options. You also need the CASE(MIXED) binder option if the entry-name 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 a UNIX file system.

For more information about DLLs, including full sample assembler DLL routines, see Chapter 4, “Building and using dynamic link libraries (DLLs),” on page 37. The following example illustrates an invocation of the CEEPCALL macro to call the routine named Bif1 with no parameters:
Figure 102: Example calling routine named Bif5 with no parameters:

The following example illustrates an invocation of the CEEPCALL macro to call the routine named Bif5 passing 5 integer parameters:

Figure 103: Example calling routine named Bif5 passing 5 integer parameters:

**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 constants, 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 is created for the imported data.

**Diagram:**

```
<table>
<thead>
<tr>
<th>Label</th>
<th>CEEPDDA</th>
<th>Data Name</th>
<th>Scope</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional label beginning in column 1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
**dataname**
Specifies the name of the data item. It is case sensitive and can be up to 255 characters in length. This entry 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 link-edit, 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 Chapter 4, “Building and using dynamic link libraries (DLLs),” on page 37.

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:

```assembly
CEEPDDA DllVar,SCOPE=EXPORT
DC      A(123)
CEEPDDA END
CEEPDDA DllStr,SCOPE=EXPORT
DC      C'Hello World'
DC      X'00'
CEEPDDA END
```

The following example illustrates how to import the variable named Biv1 into Assembler.

```assembly
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)  \---  CEEPLDA  \---  dataname  \---  REG=  \---  register
```

**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.

**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 link-edit, 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 Chapter 4, “Building and using dynamic link libraries (DLLs),” on page 37.

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.

```
* 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
```

**Example of assembler main routine**

Figure 104 on page 421 shows a simple assembler main routine. In the example, the Language Environment environment is established, a message showing control is received in the routine, and the Language Environment environment terminates with a zero return code passed in R15 to the invoker.

If you write an assembler main routine, nominate the routine as a load module entry point using the END statement, as Figure 104 on page 421 shows. Otherwise, you must explicitly declare the routine as the entry point at link-edit time.

```
*COMPILATION UNIT: LEASMMN
* ===================================================================
*        A simple main assembler routine that brings up the
*        LE/370 environment, prints a message in the main routine,
*        and returns with a return code of 0, modifier of 0.
* ===================================================================
MAIN     CEEENTRY PPA=MAINPPA
*        Invoke CEEMOUT to issue a message for us
*        CALL CEEMOUT,(STRING,DEST,0),VL   Omitted feedback code
*        Terminate the LE/370 environment and return to the caller
*        CEETERM  RC=0,MODIFIER=0
* ===================================================================
*        CONSTANTS AND WORKAREAS
* ===================================================================
* DEST     DC    F'2'
STRING   DC    Y(STRLEN)
STRBEGIN DC    C'In the main routine'
STRLEN   EQU   +-STRBEGIN
MAINPPA  CEEPPA  ,                Constants describing the code block
CEEEDA   CEEPPA  ,                Mapping of the dynamic save area
CEECAA   CEEPPA  ,                Mapping of the common anchor area
END MAIN   MAIN     Nominate MAIN as the entry point
```

Figure 104: Example of a simple main assembler routine
Example of an assembler main calling an assembler subroutine

Following is a simple assembler main routine that calls the DISPARM subroutine shown in “DSPARM subroutine example” on page 422.

+COMPILATION UNIT: LEASMSB
*====================================================================*
* A simple main assembler routine brings up Language
* Environment, calls a subroutine, and returns with
* a return code of 0.
*====================================================================*
SUBXMP CEEENTRY PPA=XMPPA,AUTO=WORKSIZE
USING WORKAREA,R13

*====================================================================*
* Invoke CEEMOUT to issue the greeting message
* CALL CEEMOUT,(HELLOMSG,DEST,FBCODE),VL,MF=(E,CEEMOUT)
* No plist to DISPARM, so zero R1. Then call it.
* SR R01,R01
CALL DISPARM
* Invoke CEEMOUT to issue the farewell message
* CALL CEEMOUT,(BYEMSG,DEST,FBCODE),VL,MF=(E,CEEMOUT)
* Terminate Language Environment and return to the caller
* CEETERM RC=0

*====================================================================*
* CONSTANTS
*====================================================================*
HELLOMSG DC Y(HELLOEND-HELLOSTR)
HELLOSTR DC C'Hello from the sub example.'
HELLOEND EQU *

BYEMSG DC Y(BYEEND-BYESTART)
BYESTART DC C'Terminating the sub example.'
BYEEND EQU *

DEST DC F'2' Destination is the LE message file
XMPPPA CEEPPA , Constants describing the code block
* The Workarea and DSA
WORKAREA DSECT
ORG *+CEEDSAZ Leave space for the DSA fixed part
CALLMOUT CALL ,(,,),VL,MF=L 3-argument parameter list
FBCODE DS 3F Space for a 12-byte feedback code

* DS OD
WORKSIZE EQU *-WORKAREA
CEEDSA , Mapping of the dynamic save area
CEECAA , Mapping of the common anchor area
R01 EQU 1
R13 EQU 13
END SUBXMP Nominate SUBXMP as the entry point

DSPARM subroutine example

+COMPILATION UNIT: LEASMPRM
*====================================================================*
* Shows an assembler subroutine that displays inbound parameters and returns.
*====================================================================*
* DISPARM CEEENTRY PPA=PARMPPA,AUTO=WORKSIZE,MAIN=NO
* USING WORKAREA,R13
* * Invoke CEE3PRM to retrieve the command parameters for us
* CALL CEE3PRM,(CHARPARM,FBCODE),VL,ML=(E,CALL3PRM)
* * Check the feedback code from CEE3PRM to see if everything worked.
* CLC FBCODE(8),CEE000
* BE GOT_PARM
* * Invoke CEEMOUT to issue the error message for us
* CALL CEEMOUT,(BADFBC,DEST,FBCODE),VL,ML=(E,CALLMOUT)
* B GO_HOME Time to go....
* 
* GOT_PARM DS 0H
* * See if the parm string is blank.
* CLC CHARPARM(80),=CL80' ' Is the parm empty?
* BNE DISPLAY_PARM No. Print it out.
* * Invoke CEEMOUT to issue the error message for us
* CALL CEEMOUT,(NOPARM,DEST,FBCODE),VL,ML=(E,CALLMOUT)
* B GO_HOME Time to go....
* 
* DISPLAY_PARM DS 0H
* * Set up the plist to CEEMOUT to display the parm.
* LA R02,80 Get the size of the string
* STH R02,BUFFSIZE Save it for the len-preixed string
* * Invoke CEEMOUT to display the parm string for us
* CALL CEEMOUT,(BUFFSIZE,DEST,FBCODE),VL,ML=(E,CALLMOUT)
* * Return to the caller
* 
* GO_HOME DS 0H
* 
* CEEETERM RC=0
* 
* ====================================================================
*              CONSTANTS
* ====================================================================
* DEST DC F'2' Destination is the LE message file
* CEE000 DS 3F'0' Success feedback code
* 
* BADFBC DC Y(BADFEND-BADFBCSTR)
* BADFBCSTR DC C'Feedback code from CEE3PRM was nonzero.'
* BADFEND EQU *
* 
* NOPARM DC Y(NO𝘢PRMEND-NOPARMSTR)
* NOPARMSTR DC C'No user parm was passed to the application.'
* NOPRMEND EQU *
* 
* PARMPPA CEEPPA , Constants describing the code block
* ====================================================================
* The Workarea and DSA
* ====================================================================
* WORKAREA DSECT ORG **CEEDSASZ Leave space for the DSA fixed part
* CALL3PRM CALL ,,(,),VL,ML=(E,CALLMOUT) 2-argument parameter list
* CALLMOUT CALL ,,(,),VL,ML=(E,CALLMOUT) 3-argument parameter list
* FBCODE DS 3F Space for a 12-byte feedback code
* 
* BUFFSIZE DS H Halfword prefix for following string
* CHARPARM DS CL255 80-byte buffer
* 
* DS 0D WORKSIZE EQU **-WORKAREA
* CEDDSA , Mapping of the dynamic save area
* CEECAA , Mapping of the common anchor area

Assembler considerations
**Invoking callable services from assembler routines**

A Language Environment-conforming assembler routine called by C should not invoke a z/OS UNIX API. The interface to a callable service is the same as the interface previously described for assembler routines. An example of calling the CEEGTST (Get Heap Storage) callable service is shown in Figure 105 on page 424.

A X'80000000' placed in the last parameter address slot indicates that the fc (feedback code) parameter is omitted.

```assembly
*   R12 = A(CAA)
*   R13 = DSA
*   This example is non-reentrant.
*   LA   R1,PLIST
   L    R15,=V(CEEGTST)
   BALR R14,R15:
   ┇
   PLIST    DS   0D
   DC   A(HEAP_ID)
   DC   A(SIZE)
   DC   A(ADDR)
   DC   A(X'80000000')
HEAP_ID  DC   F'0'                     Heap ID for the user
SIZE     DC   F'256'                   Size of storage to allocate
ADDR     DC   F'0'                     Address of allocated storage
```

_Figure 105: Sample invocation of a callable service from assembler_

**System Services available to assembler routines**

Language Environment provides a number of services that the host system typically provides. Each of these system-provided services belongs to one of three categories, depending on whether it can and ought to be used in Language Environment:

- The system-provided service can be used, but you must manage the resource; examples are ENQ and DEQ.
- The system-provided service can, but should not be used. The system-provided service might not have the desired effect. For example, instead of using GETMAIN and FREEMAIN, use the Language Environment dynamic storage callable services.
- The system-provided service should not be used. If you use this service, it directly interferes with the Language Environment environment. For example, any ESTAE or ESPIE that you issue interferes with Language Environment condition handling.

Whenever possible, non-Language Environment-conforming assembler routines should use the equivalent Language Environment services. A list of the equivalent services is provided in Table 66 on page 425.
### Table 66: Equivalent host services provided by Language Environment.

<table>
<thead>
<tr>
<th>Host service</th>
<th>Language Environment equivalent</th>
<th>Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABEND</td>
<td>Call CEESGL with a severity 4 condition, call CEE3ABD, or have the assembler user exit request an abend at termination.</td>
<td>Host services can, but should not, be used. Use of equivalent Language Environment services is advised. ABEND can be used as a last resort.</td>
</tr>
<tr>
<td>ATTACH/DETACH/CHAP¹</td>
<td>No equivalent Language Environment function.</td>
<td>These services can be used.</td>
</tr>
<tr>
<td>ENQ/DEQ</td>
<td>No equivalent Language Environment function.</td>
<td>These services can be used.</td>
</tr>
<tr>
<td>(E)STAE/(E)SPIE/ SETRP/ STAX</td>
<td>Use Language Environment condition management callable services: CEEHDLR, CEEHDLU, and CEESGL.</td>
<td>Host services should not be used; instances should be changed to use Language Environment condition management callable services. Otherwise, unpredictable results may occur.</td>
</tr>
<tr>
<td>EXEC CICS LOAD/DELETE</td>
<td>Use the Language Environment CEEFETCH assembler macro (see “CEEFETCH macro — Dynamically load a routine” on page 409).</td>
<td>Host services can be used, but you must manage the loaded routines.</td>
</tr>
<tr>
<td>EXEC CICS XCTL/LINK</td>
<td>No equivalent Language Environment function.</td>
<td>These services can be used.</td>
</tr>
<tr>
<td>GETMAIN/FREEMAIN</td>
<td>For automatic storage (block-related), use Language Environment’s stack storage. For non-block-related storage (that is, the storage persists beyond the current activation), use Language Environment heap storage.</td>
<td>Host services can, but should not, be used. Use of equivalent Language Environment storage management services is advised. Any heap storage allocated by Language Environment will automatically be freed at termination.</td>
</tr>
<tr>
<td>EXEC CICS GETMAIN/ EXEC CICS FREEMAIN</td>
<td>Use the Language Environment CEEFETCH assembler macro (see “CEEFETCH macro — Dynamically load a routine” on page 409).</td>
<td>If you are introducing a new language into the environment, host services must not be used. The new language is not properly initialized. If you are not introducing a new language into the environment, host services can be used. However, you must manage the loaded routines.</td>
</tr>
<tr>
<td>OPEN/CLOSE GET/PUT READ/ WRITE</td>
<td>No equivalent Language Environment function.</td>
<td>Host services can be used.</td>
</tr>
<tr>
<td>PC (Program Call instruction)</td>
<td>High level language call statements, such as assembler BALR/BASSM.</td>
<td>Not supported by Language Environment.</td>
</tr>
<tr>
<td>SNAP</td>
<td>Call CEE3DMP.</td>
<td>This service can be used.</td>
</tr>
<tr>
<td>STIMER²</td>
<td>No equivalent Language Environment function.</td>
<td>This service can be used.</td>
</tr>
<tr>
<td>TIME</td>
<td>Call Language Environment date and time services.</td>
<td>This service can be used.</td>
</tr>
<tr>
<td>SVC LINK</td>
<td>No equivalent Language Environment function</td>
<td>This service can be used. For compatibility, Language Environment supports the LINK boundary crossing and treats it as a new enclave.</td>
</tr>
<tr>
<td>WAIT/POST/EVENTS³</td>
<td>No equivalent Language Environment function.</td>
<td>Host services can be used.</td>
</tr>
</tbody>
</table>
Table 66: Equivalent host services provided by Language Environment. (continued)

<table>
<thead>
<tr>
<th>Host service</th>
<th>Language Environment equivalent</th>
<th>Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td>WTO</td>
<td>Call CEEMOUT. This writes to the error log or the terminal.</td>
<td>Host services can be used.</td>
</tr>
<tr>
<td>XCTL</td>
<td>No equivalent Language Environment function.</td>
<td>Host services can, but should not, be used.</td>
</tr>
</tbody>
</table>

Notes:
1. When running with POSIX(ON), use the POSIX functions pthread_create and pthread_exit in place of the host system functions ATTACH and DETACH. You cannot use ATTACH, DETACH, or CHAP when running a PL/I multitasking application.
2. When running with POSIX(ON), use the C functions ALARM and SLEEP in place of the host system function STIMER.
3. When running with POSIX(ON), use the POSIX functions pthread_mutex_lock and pthread_mutex_unlock in place of the host system functions WAIT and POST.

Using the ATTACH macro

Figure 106 on page 426 illustrates the concept of performing an OS ATTACH to a C, C++, nonmultitasking PL/I, or COBOL program, and thus establishing a separate Language Environment runtime environment. For each ATTACH to a Language Environment-conforming routine, another Language Environment runtime environment is added to the MVS address space. In COBOL, this is called multitasking; COBOL RES multitasking is supported only when all of the COBOL programs are compiled with Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370 (not with OS/VS COBOL or VS COBOL II).

Figure 106: Issuing an ATTACH to Language Environment-conforming routines

When running with POSIX(ON), use the POSIX function pthread_create in place of OS ATTACH to create a new thread.

Currently, each Language Environment environment supports one process. Within the process each enclave supports a single thread.
To make best use of the ATTACH macro, you need to be aware of:

- Whether you are using POSIX(ON) in a multithread environment. You must not use the ATTACH macro in this case. If you are running a PL/I multitasking application, you cannot use the ATTACH macro.
- Whether the Language Environment environments share any resources.
- The MVS affinity aspects of each routine. For example, if you OPEN a file in one TCB, you must CLOSE it in the same TCB.
- The concurrency aspects of each routine. For example, you must ensure that two routines do not attempt to make contradictory or destructive changes to a data base.
- The termination order of all routines, particularly those in a new Language Environment environment.
- The compiler options and link-edit options when using COBOL.

Sharing resources

Unless you indicate otherwise, the environments share the same user files and message file. To avoid conflicting use of shared resources, you should specify different ddnames using the MSGFILE runtime option, and create distinctive user files for each environment.

Note: The ENQ suboption of the MSGFILE runtime option can be used to provide serialization around writes to the Message File. See z/OS Language Environment Programming Reference for more information.

z/OS affinity aspects

z/OS calls certain pairs of commands or procedures affinity aspects, and requires each member of the pair to be issued from the same TCB.

Some examples are:

OPEN/CLOSE
If you OPEN a file in one TCB, you must CLOSE it in the same TCB. You can process the file in as many environments as you wish; you simply need to open and close it in a single TCB.

LOAD/DELETE
GETMAIN/FREEMAIN
Language Environment heap storage, including storage associated with a call to the CEECRHP callable service, will have affinity to the TCB which created the heap. All get and free requests must be done from this single TCB.

A less obvious example is a Db2 table. You can update a Db2 table only in the same TCB in which it was created.

Concurrency aspects

Concurrency aspects include which routines have access to shared resources, and the timing of changes to those resources. Because Language Environment does not provide services to lock files, to serialize access to shared resources, or to synchronize changes to shared resources, you must manage the concurrency aspects of your environments.

Termination order

Language Environment does not coordinate termination order between multiple environments. Your routines are exited properly if you adhere to the hierarchical structure created by the TCB structure.

COBOL considerations

To run COBOL programs in more than one task with Language Environment, the COBOL programs must be Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370. Running VS COBOL II programs in more than one task is not supported. When you use COBOL programs in more than one task, it is recommended that the COBOL programs be compiled with the RENT compiler option, and that the load modules be linked as REUS and RENT.
If a COBOL program running in one task dynamically calls a COBOL program that has already been dynamically called from another task, then the called program must be:

- Compiled with the RENT compiler option, or
- Compiled with the NORENT compiler option and link-edited with the NORENT and NOREUS linkage editor options.

Each copy of a COBOL program in each task will have its own unique copy of WORKING-Storage; you cannot share WORKING-STORAGE between tasks.

For example (see Figure 107 on page 428), if a COBOL program calls an assembler program, which starts a new Subtask B, and COBOL program CBL3 in the new subtask dynamically calls COBOL program CBL2 which was previously dynamically called in the main task, then CBL2 must not be link-edited with the RENT or REUS link-edit options unless it is compiled with the RENT compiler option.

```
MAIN     .
task     .  Subtask B
  CBL1     .
     |   .
     V   .
CBL2 <------.------------
     |   .
     V   .
  ASMA----(attach)---->CBL3
```

*Figure 107: A dynamically-called COBOL program that dynamically calls another COBOL program*

**Using the SVC LINK macro**

If you issue an SVC LINK, a new enclave is created. See “Creating child enclaves using SVC LINK” on page 471 for more information about nested enclaves created using the SVC LINK macro.
Chapter 30. Using preinitialization services

You can use preinitialization to enhance the performance of your application. Preinitialization 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.

The Language Environment-supplied routine, CEEPIPI, provides the interface for preinitialized routines. Using CEEPIPI, you can initialize an environment, invoke applications, terminate an environment, and add an entry to the Preinitialization table (PreInit table). (The PreInit table contains the names and entry point addresses of routines that can be executed in the preinitialized environment.)

Reentrancy considerations for a preinitialized environment, XPLINK considerations, user exit invocation, stop semantics, service routines, and an example of CEEPIPI invocation are described.

Before the introduction of a common runtime environment, introduced with Language Environment, some of the individual languages had their own form of preinitialization. This older form of preinitialization is supported by Language Environment, but it is not strategic. The following is a list of these older forms of preinitialization and some considerations for their use:

- **C**
  Language Environment supports the prior form of C preinitialization, through the use of an extended parameter list. For more information about this interface, see *z/OS XL C/C++ Programming Guide*.

- **C++**
  There is no prior form of preinitialization for C++.

- **COBOL**
  Language Environment supports the prior form of COBOL preinitialization, RTEREUS, ILBOSTP0, and IGZERRE. For more information about these interfaces, see the Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733). This prior form of COBOL preinitialization cannot be used at the same time that Language Environment preinitialization is used.

- **Fortran**
  There is no prior form of preinitialization for Fortran.

- **PL/I**
  Language Environment supports the prior form of PL/I preinitialization, through the use of an Extended Parameter List. For more information about this interface, see the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735). This prior form of PL/I preinitialization does not support PL/I multitasking applications.

**Note:** The ILBOSTP0 and IGZERRE methods of preinitialization are not supported with Enterprise COBOL v5 or later. If you are using Enterprise COBOL v5 or later, the recommended form of preinitialization is CEEPIPI.

Using preinitialization

From a non-Language Environment-conforming driver (such as assembler) you can use Language Environment preinitialization facilities to create and initialize a common runtime environment, execute applications written in a Language Environment-conforming HLL multiple times within the preinitialized environment, and terminate the preinitialized environment. Language Environment provides a preinitialized interface to perform these tasks.
In the preinitialized environment, the first routine to execute can be treated as either the main routine or a subroutine of that execution instance. Language Environment provides support for both of these types of preinitialized routines:

- Executing one main routine multiple times
- Executing subroutines multiple times

Language Environment preinitialization 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.

The interface for preinitialized routines is a loadable routine called CEEPIPI. This routine is loaded as an RMODE(24) / AMODE(ANY) routine and returns in the AMODE of its caller when the request is satisfied.

CEEPIPI handles the requests and provides services for environment initialization, application invocation, and environment termination. All requests for services by CEEPIPI must be made from a non-Language Environment environment. ("Preinitialization interface" on page 435 contains a detailed description and information about how to invoke each of these services.) The parameter list for CEEPIPI is an OS standard linkage parameter list. Each request to CEEPIPI is identified by a function code that describes the CEEPIPI service and that is the first parameter in the parameter list. The function code is a fullword integer (for example, 1 = init_main, 2 = call_main).

The preinitialization services offered under Language Environment are listed in Table 68 on page 435. Preinitialization services do not support PL/I multitasking applications.

An example assembler program in “An example program invocation of CEEPIPI” on page 466 illustrates invocation of CEEPIPI for the function codes init_sub, call_sub, and term.

Using the PreInit table

Language Environment uses the PreInit table to identify the routines that are candidates for execution in the preinitialized environment, as well as optionally to load the routine when it is called. It is possible to have an empty PreInit table with no entries. The PreInit table contains the names and the entry point addresses of each routine that can be executed within the preinitialized environment. Candidate routines can be present in the table when the init_main or init_sub functions are invoked, or can be added to the table using (add_entry).

When the entry point address is supplied either as an entry in the initial PreInit table provided with initialization functions, or as specified on the add_entry function, the high order bit of the address must be set to indicate the addressing mode for the routine. If the high order bit is OFF the routine is called in 24 bit addressing mode and the address must be a valid 24 bit address. If the high order bit is ON the routine is called in 31 bit addressing mode and the address must be a valid 31 bit address.

C considerations

C routines that are the target of (call_main) or (call_sub) must be z/OS C routines.

- C main routines must be initialized with (init_main).
- C routines that are the target of (call_main) must contain a main().

C++ considerations

The preinitialization routines (call_main) or (call_sub) can support C++ applications.

- C++ main routines must be initialized with (init_main).
- C++ routines that are the target of (call_main) must contain a main().
COBOL considerations

COBOL programs that are the target of (call_main) or (call_sub) must be Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370 programs.

Fortran considerations

Fortran routines cannot be the target of a CEEPIPI call.

PL/I considerations

PL/I routines that are the target of (call_main) or (call_sub) must be Enterprise PL/I for z/OS or PL/I for MVS & VM routines. OS PL/I Version 1 and OS PL/I Version 2 routines can run in the preinitialized environment only when called from PL/I routines that are the target of (call_main) or (call_sub).

Macros that generate the PreInit table

Language Environment provides the following assembler macros to generate the PreInit table for you: CEEXPIT, CEEXPITY, and CEEXPITS.

CEEXPIT

CEEXPIT generates a header for the PreInit table.

```
CEEXPIT
```

**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 (init_main) or a (init_sub) call.

**NOSTOR=ABEND**

Indicates that the system is to issue an abend if it cannot obtain storage for the preinitialization environment. This is the default.

**NOSTOR=RC**

Indicates that the system is to issue a return code if it cannot obtain storage for the preinitialization environment.

**STKPROT=NO**

Indicates that the preinitialized subroutine environment that was created will not be enabled for STACKPROTECT. This is the default. It has no effect for a preinitialized main environment.

**STKPROT=YES**

Indicates that the preinitialized subroutine environment that was created will be enabled for STACKPROTECT. It has no effect for a preinitialized main environment.

CEEXPITY

CEEXPITY generates an entry within the PreInit table.

```
CEEXPITY
```

**name**

The first eight characters of the load name of a routine that can be invoked within the Language Environment preinitialized environment.
entry_point

The address of the load module that is to be invoked, or 0, to indicate that the module is to be
dynamically loaded.

The high-order bit of the entry_point address must be set to indicate the addressing mode for the routine.
If the high-order bit is OFF, the routine is called in 24 bit addressing mode and the address must be a valid
24 bit address. If the high-order bit is ON, the routine is called in 31 bit addressing mode and the address
must be a valid 31 bit address.

You have the option of specifying either, both, or neither of the parameters:
• If name is omitted and entry_point is present, the comma must be present.
• If both parameters are omitted, the entry is a candidate for assignment to the PreInit table by a call to
  (add_entry).
• If both parameters are present, name is ignored and entry_point is used as the start of the routine.

Each invocation of the CEEXPITY macro generates a row in the PreInit table. The first entry is row 0, the
second is row 1, and so on.

CEEXPITS

CEEXPITS identifies the end of the PreInit table. This macro has no parameters.

Reentrancy considerations

You can make multiple calls to main routines by invoking CEEPIPI services and making multiple requests
from a single PreInit table. In general, you should specify only reentrant routines for multiple invocations,
or you might get unexpected results.

For example, if you have a reentrant C main program that is invoked using (call_main) and that uses
external variables, then when your routine is invoked again, the external variables are re-initialized.
Multiple executions of a reentrant main routine are not influenced by a previous execution of the same
routine.

However, if you have a nonreentrant C main program that is invoked using (call_main) and that uses
external variables, then when your routine is invoked again, the external variables can potentially contain
last-used values. Local variables (those contained in the object code itself) might also contain last-used
values. If main routines are allowed to execute multiple times, a given execution of a routine can influence
subsequent executions of the same routine.

If you are calling init_sub, init_sub_dp, or add_entry for C/C++, the routines can either be naturally
reentrant or may be compiled RENT and made reentrant by using the z/OS C Prelinker Utility. If the
subroutine is made reentrant using the z/OS C Prelinker Utility, multiple instances of the same subroutine
are influenced by the previous instance of the same subroutine.

If you have a nonreentrant COBOL program that is invoked using (call_main), condition IGZ0044S is
signaled when the routine is invoked again.

PreInit XPLINK considerations

Language Environment preinitialization services (PreInit) support programs that have been compiled
XPLINK. Specifically, it allows programs and subroutines that have been compiled XPLINK to be defined
in the PreInit table. The following guidelines are provided for this new option:
• XPLINK CEEPIPI subroutines must be fetchable. For C programs, this is done using the #pragma linkage
  (fetchable) statement. For more details on fetchable subroutines, refer to the documentation on fetch() in
  z/OS XL C/C++ Runtime Library Reference.
• Non-XPLINK PreInit programs can run in an XPLINK PreInit environment, but there may be performance degradation since non-XPLINK programs will be required to execute linkage-switching glue code. If possible, consider having separate PreInit environments for running XPLINK and non-XPLINK programs.

• If a PreInit environment has been initialized as a non-XPLINK environment and either the main() function is XPLINK or the XPLINK(ON) runtime option has been specified, then the PreInit environment will be rebuilt as an XPLINK environment. This is a one-time occurrence that can not be undone.

Creating an XPLINK environment versus a non-XPLINK environment

When initializing a PreInit environment, you can select to create an XPLINK or a non-XPLINK environment. There are four methods used to initialize a PreInit environment; init_main, init_main_dp, init_sub, and init_sub_dp. In each case, a token of the preinitialized environment is passed back to the customer PreInit driver program. This token ID is used and passed as input when executing PreInit programs. The following rules will determine if the initialized PreInit environment will be XPLINK or non-XPLINK. You can make a one-time dynamic change in the PreInit environment from non-XPLINK to XPLINK by using (call_main) to an XPLINK main().

init_main: (Input: PreInit table pointer, no runtime options are passed as input)

• If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.

• If the first program in the PreInit table is a non-XPLINK program, then a non-XPLINK environment will be initialized.

• If the PreInit table is empty at initialization time, then a non-XPLINK environment will be initialized.

init_main_dp: (Input: PreInit table pointer, no runtime options are passed as input)

• If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.

• If the first program in the PreInit table is a non-XPLINK program, then a non-XPLINK environment will be initialized.

• If the PreInit table is empty at initialization time, then a non-XPLINK environment will be initialized.

init_sub: (Input: PreInit table pointer, and runtime options)

• If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.

• If the runtime options are passed as input and the XPLINK option is specified as XPLINK(ON), then an XPLINK environment will be initialized.

• If neither of the above are true (the first program in the customer PreInit table is a non-XPLINK program and the XPLINK runtime option is off or not specified), then a non-XPLINK environment will be initialized.

Note:

1. The runtime options you specify will apply to all of the subroutines that are called by (call_sub) function. This includes options such as XPLINK. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.

2. If this is a non-XPLINK sub environment, then do not allow an XPLINK subroutine to be added to the table.

init_sub_dp: (Input: PreInit table pointer, and runtime options)

• If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.

• If the runtime options are passed as input and the XPLINK option is specified as XPLINK(ON), then an XPLINK environment will be initialized.

• If neither of the above are true (the first program in the customer PreInit table is a non-XPLINK program and the XPLINK runtime option is off or not specified), then a non-XPLINK environment will be initialized.
Note: The runtime options you specify will apply to all of the subroutines that are called by (call_sub) function. This includes options such as XPLINK. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.

User exit invocation

User exits are invoked for initialization and termination during calls to CEEPIPI as shown in Table 67 on page 434.

<table>
<thead>
<tr>
<th>Function</th>
<th>When invoked</th>
</tr>
</thead>
</table>
| Assembler user exit for first enclave initialization | • (init_sub)  
• (init_sub_dp)  
• (call_main)  
• (call_sub) or (call_sub_addr) or (call_sub_addr) ended with stop semantics (see “Stop semantics” on page 434) |
| HLL user exit | • (init_sub)  
• (init_sub_dp)  
• (call_main)  
• (call_sub) or (call_sub_addr) or (call_sub_addr) ended with stop semantics |
| Catexit() functions | • (call_main)  
• (call_sub) or (call_sub_addr), which ended stop semantics.  
• (term) for environment created with (init_sub) or (init_sub_dp), if the last (call_sub) or (call_sub_addr) did not end with stop semantics |
| Assembler user exit for first enclave termination | • (call_main)  
• (call_sub) or (call_sub_addr), which ended stop semantics  
• (term) for environment created with (init_sub) or (init_sub_addr) if the last (call_sub) or (call_sub_addr) did not end with stop semantics |
| Assembler user exit for process termination | • (term) |

For main environments, the CEEBXITA assembler user exit and CEEBINT HLL user exit that are used with the environment are taken from the main routine being called.

For sub environments, the CEEBXITA assembler user exit and CEEBINT HLL user exit that are used with the environment are taken from the first entry in the PreInit table. Any occurrences of CEEBXITA or CEEBINT in any other PreInit table entries, or in load modules used for call_sub_addr-type calls, are ignored.

See Chapter 28, “Using runtime user exits,” on page 373 for more information about user exits.

Stop semantics

When one of the following is issued within the preinitialized environment for subroutines:
• C exit(), abort(), or signal handling function specifying a normal or abnormal termination  
• COBOL STOP RUN statement  
• PL/I STOP or EXIT
or when an unhandled condition causes termination of the (only) thread, the logical enclave is terminated. The process level of the environment is retained. Language Environment does not delete those entries that were loaded explicitly by Language Environment during the preinitialization processing.

⚠️ **Attention:** If the first entry in the PreInit table is either different or deleted from when the enclave was last initialized, the assembler user exit (CEEBXITA), HLL user exit (CEEBINT), or programmer default runtime options (CEEUOPT) used during either an enclave reinitialization or enclave termination will either be different or not available. This will result in unpredictable results. Therefore, when using PreInit subroutine environments and in order to keep consistent enclave initialization and termination behavior, users need to ensure the first valid entry in the PreInit table does not change, especially when it contains the aforementioned external references.

### Preinitialization interface

This section describes how to invoke the PreInit interface, CEEPIPI, 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
- Access to the CAA user word

The PreInit services offered under Language Environment using CEEPIPI are listed in the following tables:

- Table 68 on page 435
- Table 69 on page 436
- Table 70 on page 436
- Table 71 on page 436
- Table 72 on page 436
- Table 73 on page 436
- Table 74 on page 436
- Table 75 on page 436

For Initialization:

<table>
<thead>
<tr>
<th>Function code</th>
<th>Integer value</th>
<th>Service performed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>init_main</code></td>
<td>1</td>
<td>Create and initialize an environment for multiple executions of main routines.</td>
</tr>
<tr>
<td><code>init_main_dp</code></td>
<td>19</td>
<td>Create and initialize an environment for multiple executions of main routines.</td>
</tr>
<tr>
<td><code>init_sub</code></td>
<td>3</td>
<td>Create and initialize an environment for multiple executions of subroutines.</td>
</tr>
<tr>
<td><code>init_sub_dp</code></td>
<td>9</td>
<td>Create and initialize an environment for multiple executions of subroutines.</td>
</tr>
</tbody>
</table>

For application invocation:
Table 69: Preinitialization services accessed using CEEPIPI for application invocation.

<table>
<thead>
<tr>
<th>Function code</th>
<th>Integer value</th>
<th>Service performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>call_main</td>
<td>2</td>
<td>Invoke a main routine within an already initialized environment.</td>
</tr>
<tr>
<td>call_sub</td>
<td>4</td>
<td>Invoke a subroutine within an already initialized environment.</td>
</tr>
<tr>
<td>start_seq</td>
<td>7</td>
<td>Start a sequence of uninterruptable calls to a number of subroutines.</td>
</tr>
<tr>
<td>call_sub_addr</td>
<td>10</td>
<td>Invoke a subroutine by address within an already initialized environment.</td>
</tr>
</tbody>
</table>

For termination:

Table 70: Preinitialization services accessed using CEEPIPI for termination.

<table>
<thead>
<tr>
<th>Function code</th>
<th>Integer value</th>
<th>Service performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>5</td>
<td>Explicitly terminate the environment without executing a user routine.</td>
</tr>
<tr>
<td>end_seq</td>
<td>8</td>
<td>Terminate a sequence of uninterruptable calls to a number of subroutines.</td>
</tr>
</tbody>
</table>

For the addition of entries to the PreInit table:

Table 71: Preinitialization services accessed using CEEPIPI for the addition of an entry to the PreInit table.

<table>
<thead>
<tr>
<th>Function code</th>
<th>Integer value</th>
<th>Service performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_entry</td>
<td>6</td>
<td>Dynamically add a candidate routine to execute within the preinitialized environment.</td>
</tr>
</tbody>
</table>

For the deletion of entries from the PreInit table:

Table 72: Preinitialization services accessed using CEEPIPI for the deletion of an entry to the PreInit table.

<table>
<thead>
<tr>
<th>Function code</th>
<th>Integer value</th>
<th>Service performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete_entry</td>
<td>11</td>
<td>Delete an entry from the PreInit table, making it available for subsequent add_entry functions.</td>
</tr>
</tbody>
</table>

For the identification of a PreInit table entry:

Table 73: Preinitialization services accessed using CEEPIPI for the identification of a PreInit table entry.

<table>
<thead>
<tr>
<th>Function code</th>
<th>Integer value</th>
<th>Service performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>identify_entry</td>
<td>13</td>
<td>Identify the programming language of an entry in the PreInit table.</td>
</tr>
<tr>
<td>identify_attributes</td>
<td>16</td>
<td>Identify the attributes of an entry in the PreInit table.</td>
</tr>
</tbody>
</table>

For the identification of the environment:

Table 74: Preinitialization services accessed using CEEPIPI for the addition of an entry to the PreInit table.

<table>
<thead>
<tr>
<th>Function code</th>
<th>Integer value</th>
<th>Service performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>identify_environment</td>
<td>15</td>
<td>Identify the environment that was preinitialized.</td>
</tr>
</tbody>
</table>

For the access to the CAA user word:

Table 75: Preinitialization services accessed using CEEPIPI for access to the CAA user word.

<table>
<thead>
<tr>
<th>Function code</th>
<th>Integer value</th>
<th>Service performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_user_word</td>
<td>17</td>
<td>Set value to be used to initialize CAA user word.</td>
</tr>
<tr>
<td>get_user_word</td>
<td>18</td>
<td>Get value to be used to initialize CAA user word.</td>
</tr>
</tbody>
</table>
Initialization

Language Environment supports four forms of preinitialized environments. The first supports the execution of main routines. The second is a special form of the first, that allows multiple preinitialized environments, for executing main routines, to be created within the same address space. The third supports the execution of subroutines. The fourth is a special form of the third, that allows multiple preinitialized environments, for executing subroutines, to be created within the same address space.

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.

For the **main environment**, the advantages are:

- A new, pristine environment is created.
- Runtime options can be specified for each application.

and the disadvantages are:

- Poorer performance.

For the **subenvironment**, the advantages are:

- Best performance.

and the disadvantages are:

- The environment is left in whatever state the previous application left it in.
- Runtime options cannot be changed.

**(init_main) — initialize for main routines**

The invocation of this routine:

- Creates and initializes a new common runtime environment (process) that allows the execution of main routines multiple times
- Sets the environment to 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

---

**init_main (input)**

A fullword function code (integer value = 1) containing the init_main request.

**ceextbl_addr (input)**

A fullword 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 (in the LPA, saved segment, or nucleus) and dynamically loads it. Language Environment places the entry address in the corresponding slot of a Language Environment-maintained table.

Language Environment uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the entry address is zero, and the entry name is supplied, Language Environment uses the AMODE returned by the system loader. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.
Preinitialization services

**service_rtns (input)**
A fullword containing the address of the service routine vector or 0, if there is no service routine vector. See “Service routines” on page 459 for more information.

**token (output)**
A fullword containing a unique value used to represent the environment. The token should be used only as input to additional calls to CEEPIPI, 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 (in decimal) are:

- **0**
  A new environment was successfully initialized.

- **4**
  The function code is not valid.

- **8**
  All addresses in the table were not resolved. This can occur if a LOAD failure was encountered or a routine within the table was generated by a non-Language Environment-conforming HLL.

- **12**
  Storage for the preinitialization environment could not be obtained.

- **16**
  CEEPIPI was called from an active environment.

- **32**
  An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

**Usage notes**
- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the main routine being called through call_main.
- If a program in the PreInit table failed to load (return code 8), the `identify_attributes` CEEPIPI function can be used to help determine what table entry address did not resolve.

**XPLINK considerations**
- If the environment being initialized is to be an XPLINK environment then the first program in the PreInit table must be an XPLINK module.
- If there is no entry in the PreInit table or if the first module is a non-XPLINK program, a non-XPLINK environment will be initialized.
- It is possible to change the environment from a non-XPLINK to an XPLINK environment when doing a call_main. For more details, see call_main.

**(init_main_dp) — initialize for main routines (multiple environment)**
The invocation of this routine:
- Creates and initializes a new common runtime 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.
- Ensures that the environment tolerates the existence of multiple Language Environment processes or enclaves.
**Note:** Multiple main environments can be established by using `init_main_dp`, as opposed to using `init_main`, which can establish only a single environment.

---

**init_main_dp**

A fullword function code (integer value = 19) containing the `init_main_dp` request.

**ceexptbl_addr**

A fullword containing the address of the PreInit table to be used during initialization of the new environment. A user-supplied copy of the table is not altered. If an entry address is zero and the entry name is non-blank, a search is performed for the routine (in the LPA, saved segment, or nucleus) and the routine is dynamically loaded. An entry is placed in the corresponding slot of a Language Environment-maintained table.

The high-order bit of the entry address determines what AMODE to use when calling the routine. If the entry address is zero, and the entry name is supplied, the AMODE returned by the system loader is used. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

**service_rtns**

A fullword containing the address of the service routine vector or 0, if there is no service routine vector. See “Service routines” on page 459 for more information.

**token**

A fullword containing a unique value used to represent the environment. The `token` should be used only as input to additional calls to CEEPIPI, 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 (in decimal) are:

- **0**  
  A new environment was successfully initialized.

- **4**  
  The function code is not valid.

- **8**  
  All addresses in the table were not resolved. This can occur if a LOAD failure was encountered or a routine within the table was generated by a non-Language Environment-conforming HLL.

- **12**  
  Storage for the preinitialization environment could not be obtained.

- **16**  
  CEEPIPI was called from an active environment other than a CEEPIPI main_dp environment.

- **32**  
  An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

**Usage notes**

- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the main routine being called through `call_main`.

- If a program in the PreInit table failed to load (return code 8), the `(identify_attributes)` CEEPIPI function can be used to help determine what table entry address did not resolve.

- If the process ID needs to be the same for all programs called by `call_main`, the preinitialization driver program should pre-dub the task (TCB) before performing `init_main_dp`.
Preinitialization services

- MSGFILE output can be directed to either a spool or to a unique file.
- Language Environment resources are not shared across multiple environments.
- C memory files are not shared across multiple environments.
- Calling POSIX(ON) programs in an (init_main_dp) environment is not supported.

**XPLINK considerations**

- If the environment being initialized is to be an XPLINK environment then the first program in the PreInit table must be an XPLINK module.
- If there is no entry in the PreInit table or if the first module is a non-XPLINK program, a non-XPLINK environment will be initialized.
- It is possible to change the environment from a non-XPLINK to an XPLINK environment when using (call_main). For more information, see “(call_main) — invocation for main routine” on page 444.

**Nested main_dp environment considerations**

- Main_dp environments can be initialized by calling CEEPIPI(init_main_dp) from an active main_dp environment. From an active main_dp environment, nested calls to CEEPIPI can be made with a token returned from (init_main_dp) to perform certain other functions:
  - (call_main)
  - (add_entry)
  - (delete_entry)
  - (term)
  - (set_user_word)
  - (get_user_word)
  - (identify_entry)
  - (identify_environment)
  - (identify_attributes)

- Restrictions for nested main_dp environments:
  - When the calling environment has a user-provided @EXCEPRTN, the nested main_dp environment must also have a user-provided @EXCEPRTN.
  - If the user-written preinitialization driver program has established a SPIE or ESPIE routine, the nested main_dp environment must have a user-provided @EXCEPRTN.
  - All CEEPIPI calls that use a token must be made from the same TCB.
  - The INTERRUPT(ON) runtime option is not supported when using nested main_dp environments under TSO/E.
  - When the TRAP runtime option is used with nested main_dp environments, use of the TSO/E attention key is not supported.
  - If an ABEND (40XX, for example) causes the immediate ending of a nested main_dp environment without orderly Language Environment termination, the user-provided preinitialization driver program cannot be returned to. The calling main_dp environment will also end without orderly Language Environment termination.
  - If the ABTERMENC(ABEND) runtime option is in effect and an unhandled condition causes a nested main_dp environment to ABEND, Language Environment will not return to the preinitialization assembler driver program. The calling main_dp environment will also ABEND without orderly Language Environment termination. Consider using ABTERMENC(RETCODE) in nested main_dp environments.
  - If a main_dp environment which uses the TRAP(ON,SPIE) runtime option does (call_main) to a nested main_dp environment which uses TRAP(ON,NOSPIE), language environment issues an ESPIE macro to prevent program checks from being passed to any existing ESPIE routine. If this ESPIE call
must be avoided, do not call a nested main_dp environment with TRAP(ON,NOSPIE) from a main_dp environment that uses TRAP(ON,SPIE).

(init_sub) — initialize for subroutines

The invocation of this routine:

• Creates and initializes a new common runtime 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

CALL CEEPIPI (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 fullword 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 (in the LPA, saved segment, or nucleus) and dynamically loads it. Language Environment then places the entry address in the corresponding slot of a Language Environment-maintained table.

Language Environment uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the entry address is zero, and the entry name is supplied, Language Environment uses the AMODE returned by the system loader. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

service_rtns (input)
A fullword containing the address of the service routine vector. It contains 0 if there is no service routine vector. See “Service routines” on page 459 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:
1. 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.
2. If the Language Environment being initialized is a non-XPLINK environment, then all of your subroutines must be non-XPLINK subroutines.

token (output)
A fullword containing a unique value used to represent the environment. The token should be used only as input to additional calls to CEEPIPI, 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 (in decimal) are:
A new environment was successfully initialized.

The function code is not valid.

All addresses in the table were not resolved. This can occur if a LOAD failure was encountered, a routine within the table was not generated by a Language Environment-conforming HLL, or a C or PL/I routine within the table was not fetchable.

Storage for the preinitialization environment could not be obtained.

CEEPICI was called from an active environment.

An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

An entry in the PreInit table is an XPLINK subroutine and the environment is a non-XPLINK sub environment. This entry is not valid.

Usage notes

• The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the first valid entry in the PreInit table. Any occurrences of CEEBXITA, CEEBINT, and CEEUOPT in other PreInit table entries are ignored. Unpredictable results will occur if this first entry is deleted or changed.

• If a program in the PreInit table failed to load (return code 8 or 40), the identify_attributes CEEPIPI function can be used to help determine what table entry address did not resolve.

XPLINK considerations

• If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.

• If the runtime options are passed as input and the XPLINK option is specified as XPLINK(ON), then an XPLINK environment will be initialized.

• If neither of the above are true (the first program in the customer PreInit table is a non-XPLINK program and the XPLINK runtime option is off or not specified), then a non-XPLINK environment will be initialized.

Note:

1. The runtime options you specify will apply to all of the subroutines that are called by (call_sub) function. This includes options such as XPLINK. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.

2. If this is a non-XPLINK sub environment, then do not allow an XPLINK subroutine to be added to the table.

(init_sub_dp) — initialize for subroutine (multiple environment)

The invocation of this routine:

• Creates and initializes a new Language Environment process and enclave to allow 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 the environment tolerates the existence of multiple Language Environment enclaves
Preinitialization services

- Ensures that when the environment is dormant, it is immune to other Language Environment enclaves that are created or terminated

Multiple environments can be established only by using (init_sub_dp) as opposed to (init_sub), which can establish only a single environment.

\[
\begin{align*}
\text{CALL} & \quad \text{CEEPIPI} \quad (\text{init_sub_dp}, \text{ceexptbl_addr}, \text{service_rtns}, \text{runtime_opts}, \text{token}) \\
\text{runtime_opts} & \quad \text{token} 
\end{align*}
\]

\textit{init_sub_dp (input)}
A fullword function code (integer value = 9) containing the \textit{init_sub_dp} request.

\textit{ceexptbl_addr (input)}
A fullword 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 (in the LPA, saved segment, or nucleus) and dynamically loads it. Language Environment then places the entry address in the corresponding slot of a Language Environment-maintained table.

Language Environment uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the entry address is zero, and the entry name is supplied, Language Environment uses the AMODE returned by the system loader. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

\textit{service_rtns (input)}
A fullword containing the address of the service routine vector. It contains 0 if there is no service routine vector. See “Service routines” on page 459 for more information.

\textit{runtime_opts (input)}
A fixed-length 255-character string containing runtime options (see \textit{z/OS Language Environment Programming Reference} for a list of runtime options that you can specify).

\textbf{Note:}

1. 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.

2. If you want to run XPLINK routines in a PreInit sub environment, you must specify the XPLINK(ON) runtime option field when you create the sub environment by calling CEEPIPI(\textit{init_sub}). You cannot run XPLINK routines in a sub environment when runtime option XPLINK(OFF) is in effect.

\textit{token (output)}
A fullword containing a unique value used to represent the environment. The \textit{token} should be used only as input to additional calls to CEEPIPI, and should not be altered or used in any other manner.

\textbf{Return codes}
Register 15 contains a return code indicating if an environment was successfully initialized. Possible return codes (in decimal) are:

0  
A new environment was successfully initialized.

4  
The function code is not valid.

8  
All addresses in the table were not resolved. This can occur if a LOAD failure was encountered or a routine within the table was not generated by a Language Environment-conforming HLL.

12  
Storage for the preinitialization environment could not be obtained.
An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

An entry in the PreInit table is an XPLINK subroutine and the environment is a non-XPLINK sub environment. This entry is not valid.

Usage notes
• The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the first valid entry in the PreInit table. Any occurrences of CEEBXITA, CEEBINT, and CEEUOPT in other PreInit table entries are ignored. Unpredictable results will occur if this first entry is deleted or changed.
• COBOL, PL/I, and C routines must be compiled RENT to participate in this environment
• You can direct MSGFILE output to either a spool or to a unique file.
• C memory files are not shared across multiple environments.
• If the (init_sub_dp,...) interface is used to create additional environments, neither the existing environment, nor the one trying to be created can be POSIX(ON).
• If a program in the PreInit table failed to load (return code 8 or 40), the identify_attributes CEEPIPI function can be used to help determine what table entry address did not resolve.

XPLINK considerations
• If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.
• If the runtime options are passed as input and the XPLINK option is specified as XPLINK(ON), then an XPLINK environment will be initialized.
• If neither of the above are true (the first program in the customer PreInit table is a non-XPLINK program and the XPLINK runtime option is off or not specified), then a non-XPLINK environment will be initialized.

Note: The runtime options you specify apply to all of the subroutines that are called by (call_sub_dp) function. This includes options such as XPLINK. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.

Application invocation
Language Environment provides facilities to invoke either a main routine or subroutine. When invoking main routines, the environment must have been initialized using the init_main or init_main_dp function code. Similarly, when invoking subroutines, the environment must have been initialized with the init_sub or init_sub_dp function codes.

(call_main) — invocation for main routine
This invocation of CEEPIPI invokes as a main routine the routine that you specify. The common execution environment identified by token is activated before the called routine is invoked, and after the called routine returns, the environment is dormant.

At termination, the currently active HLL event handlers are driven to enforce language semantics for the termination of an application such as closing files and freeing storage. The process level is made dormant rather than terminated. The thread and enclave levels are terminated. The assembler user exit is driven with the function code for first enclave termination. (For more information about user exits, see Chapter 28, “Using runtime user exits,” on page 373.)
CALL CEEPIPI (call_main, ceexptl_index, token, runtime_opts, parm_ptr, enclave_return_code, enclave_reason_code, appl_feedback_code)

**call_main (input)**
A fullword function code (integer value = 2) containing the call_main request.

**ceexptbl_index (input)**
A fullword containing the row number within the PreInit table of the entry that should be invoked. The index starts at 0.

Each invocation of the CEEXPITY macro generates a row in the PreInit table. The first entry is row 0, the second is row 1 and so on. A call to (add_entry) to add an entry to the PreInit table also returns a row number in the ceexptbl_index parameter.

**token (input)**
A fullword with the value of the token returned by (init_main) or (init_main_dp) when the common runtime 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 fullword parameter list pointer or 0 (zero) that is placed in register 1 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 132.

**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 132.

**appl_feedback_code (output)**
A 96-bit condition token indicating why the application terminated.

**Return codes**
Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0  The environment was activated and the routine called.

4  The function code is not valid.

8  If token was initialized by (init_main) or (init_sub), CEEPIPI(call_main) was called from a Language Environment-conforming HLL.

   If token was initialized by (init_main_dp), CEEPIPI(call_main) was called from a Language Environment-conforming HLL that is not running in a (main_dp) environment, or token is already in use for another call to CEEPIPI.
The indicated environment was initialized for subroutines. No routine was executed.

The **token** is not valid.

The index points to an entry that is not valid or empty.

The index that was passed is outside the range of the table.

An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

The user return code and Language Environment return code modifier are set to zero before invoking the target routine.

**Usage notes**

- The NOEXECOPS and CBLOPTS runtime options (see *z/OS Language Environment Programming Reference*) are ignored since the parameter inbound to the application and the runtime options are separated already. Therefore, NOEXECOPS and CBLOPTS do not affect the parameter string format. See “C PLIST and EXECOPS interactions” on page 505 for more information.
- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the main routine being called. Any occurrences of CEEBXITA, CEEBINT, and CEEUOPT in other PreInit table entries are ignored.
- For more information about return codes, see “Managing return codes in Language Environment” on page 132.

**call_sub** — invocation for subroutines

This invocation of CEEPIPI invokes as a subroutine the routine that you specify. The common runtime environment identified by **token** is activated before the called routine is invoked, and after the called routine returns, the environment is dormant.

The enclave is terminated when an unhandled condition is encountered or a STOP statement is executed. (See “Stop semantics” on page 434 for more information.) However, the process level is maintained. The next call to (call_sub) initializes a new enclave.

```
call_sub (input)
A fullword function code (integer value = 4) containing the call_sub request for a subroutine.

ceexptbl_index (input)
A fullword containing the row number of the entry within the PreInit table that should be invoked; the index starts at 0.

  **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.

token (input)
A fullword with the value of the token returned when the common runtime environment is initialized. This token is initialized by the (init_sub) or (init_sub_dp). The **token** must identify a previously preinitialized environment that is not active at the time of the call. You must not alter the value of the token.
**Note:** If the token pointing to the previously preinitialized environment is a non-XPLINK environment and the subprogram to be invoked is XPLINK a Return Code of 40 will be returned because this is not valid.

**parm_ptr (input)**
A parameter list pointer or 0 (zero) that is placed in register 1 when the routine is executed.

C and C++ users need to follow the subroutine linkage convention for C/C++ — assembler ILC applications, as described in *z/OS XL C/C++ Programming Guide*.

**sub_ret_code (output)**
The subroutine return code. If the enclave is terminated due to an unhandled condition, a STOP statement, or EXIT statement (or an exit() function), this contains the enclave return code for termination.

**sub_reason_code (output)**
The subroutine reason code. This is 0 for normal subroutine returns. If the enclave is terminated due to an unhandled condition, a STOP statement, or EXIT statement (or an exit() function), this contains the enclave reason code for termination.

**sub_feedback_code (output)**
The feedback code for enclave termination. This is the CEE000 feedback code for normal subroutine returns. If the enclave is terminated due to an unhandled condition, a STOP statement, or EXIT statement (or an exit() function), this contains the enclave feedback code for termination.

**Return codes**
Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0  
The environment was activated and the routine called.

4  
The function code is not valid.

8  
CEEPIPI was called from a Language Environment-conforming HLL.

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.

40  
The subprogram was an XPLINK program and the preinitialized environment is non-XPLINK. This is not valid.

**Usage notes**
- The enclave terminates if the subroutine issues a STOP statement, EXIT statement (or an exit() function), or if there is an unhandled condition. However, the process level is not terminated. When the enclave level is terminated, any subsequent invocation creates a new enclave by using the same
runtime options used in the creation of the first enclave. Language Environment does not delete any user routines that were loaded into the PreInit table.

However, if, the first valid entry in the PreInit table is different than when the enclave was last initialized, the assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and/or programmer default runtime options (CEEUOPT) used during the enclave re-initialization might be different. PreInit subroutine initialization uses these external references only when associated with the first valid entry in the PreInit table. Therefore, when using PreInit subroutine environments and you want consistent enclave initialization behavior across the stop semantics, you need to ensure the first valid entry in the PreInit table does not change, especially when it contains the aforementioned external references.

(See “Stop semantics” on page 434.)

• Any subroutine that modifies external data cannot make assumptions about the initial state of that external data. The initial state of the external data is influenced by previous instances of the same subroutine and also by previous instances of any subroutine that caused enclave termination.

• If the first entry in the PreInit table contained a CEEBXITA, CEEBINENT or CEEUOPT when the environment was initialized and is then deleted or changed, the results of subsequent enclave re-initialization or termination is unpredictable. It is the responsibility of the user to ensure the first entry in the PreInit table does not change, especially when it contains the aforementioned external references.

(call_sub_addr) — invocation for subroutines by address

This invocation of CEEPIPI invokes a specified routine as a subroutine. The common runtime environment identified by token is activated before the called routine is invoked; after the called routine returns, the environment is dormant.

The enclave is terminated when an unhandled condition is encountered or a STOP or EXIT statement (or an exit() function) is executed. (See “Stop semantics” on page 434 for more information.) However, the process level is maintained; only the enclave level terminates.

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. The first fullword contains the entry point address.

Note:

1. If this is an XPLINK environment and the second fullword is zero, Preinitialization services will create a new function pointer to call the routine directly. The new function pointer will be returned in the second fullword.

2. If this is an XPLINK environment and the second fullword is a function pointer, the XPLINK subroutine is called directly. This fast path avoids the overhead of translating the routine address to the function pointer.

token (input)

A fullword with the value of the token returned by (init_sub) or (init_sub_dp) when the common runtime environment is initialized. The token must identify a previously preinitialized environment that is not active at the time of the call. You must not alter the value of the token.

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 parameter list pointer or 0 (zero) that is placed in register 1 when the routine is executed.
C and C++ users are advised to follow the subroutine linkage convention for C/C++ — assembler ILC applications, as described in z/OS XL C/C++ Programming Guide.

**sub_ret_code (output)**
The subroutine return code. If the enclave is terminated due to an unhandled condition or a STOP or EXIT statement (or an exit() function), this contains the enclave return code for termination.

**sub_reason_code (output)**
The subroutine reason code. This is 0 for normal subroutine returns. If the enclave is terminated due to an unhandled condition or a STOP or EXIT statement (or an exit() function), this contains the enclave reason code for termination.

**sub_feedback_code (output)**
The feedback code for enclave termination. This is the CEE000 feedback code for normal subroutine returns. If the enclave is terminated due to an unhandled condition or a STOP or EXIT statement (or an exit() function), this contains the enclave feedback code for termination.

**Return codes**
Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

- **0**
  The environment was activated and the routine called.

- **4**
  The function code is not valid.

- **8**
  CEEPIPI was called from a Language Environment-conforming HLL.

- **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.

- **40**
  The subprogram was an XPLINK program and the preinitialized environment is non-XPLINK. This is not valid.

- **41**
  Indicates the routine address could not be converted to a function descriptor.

**Usage notes**
- The enclave terminates if the subroutine issues a STOP or EXIT statement (or an exit() function), or if there is an unhandled condition. However, the process level is not terminated. When the enclave level is terminated, any subsequent invocation creates a new enclave using the same runtime options used in the creation of the first enclave. Language Environment does not delete any user routines that were loaded into the PreInit table.

  However, if, the first valid entry in the PreInit table is different than when the enclave was last initialized, the assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and/or programmer default runtime options (CEEUOPT) used during the enclave re-initialization might be different. PreInit subroutine initialization uses these external references only when associated with the first valid entry in the PreInit table. Therefore, when using PreInit subroutine environments and you want consistent enclave initialization behavior across the stop semantics, you need to ensure the first valid entry in the PreInit table does not change, especially when it contains the aforementioned external references.
(See “Stop semantics” on page 434.)

• Any subroutine that modifies external data cannot make assumptions about the initial state of that external data. The initial state of the external data is influenced by previous instances of the same subroutine and also by previous instances of any subroutine that caused enclave termination.

• C subroutines that are not naturally reentrant and C++ subroutines can be invoked using call_sub_addr only in an XPLINK environment. In a non-XPLINK environment, they must be invoked using call_sub.

• If the first entry in the PreInit table contained a CEEBXITA, CEEBINIT or CEEUOPT when the environment was initialized and is then deleted or changed, the results of subsequent enclave re-initialization or termination is unpredictable. It is the responsibility of the user to ensure the first entry in the PreInit table does not change, especially when it contains the aforementioned external references.

(\texttt{end\_seq}) — end a sequence of calls

This invocation of CEEPIPI declares that a sequence of uninterrupted calls to subroutines by this driver program has finished.

\texttt{CALL CEEPIPI (end\_seq, token)}

\texttt{end\_seq} (input)

A fullword function code (integer value = 8) containing the end_seq request

\texttt{token} (input)

A fullword with the value of the token returned by (init_sub_dp) when the common runtime environment is initialized.

The \texttt{token} must identify a previously preinitialized environment that was prepared for multiple calls by the (start_seq) call.

\textbf{Return codes}

Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0  The environment is no longer prepared for a sequence of calls.
4  The function code is not valid.
8  The indicated environment was already active; no action taken.
16  The \texttt{token} is not valid.
20  The \texttt{token} was not used in a start_seq call.

\textbf{Usage notes}

• (end_seq) can be used only in conjunction with a Language Environment environment initialized by an (init_sub_dp) function code. A return code of 4 is set for environments initialized by other than (init_sub_dp).

• Only (call_sub) or (call_sub_addr) invocations are allowed between the (start_seq) and (end_seq) calls.

• The driver program cannot cancel any STAE or ESPIE routines.

• This function can be called from an active environment if the Preinitialization environment indicated by \texttt{token} was created with the (init_sub_dp) function.
(start_seq) — start a sequence of calls

This invocation of CEEPIPI 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.

CALL CEEPIPI (start_seq, token)

start_seq (input)
A fullword function code (integer value = 7) containing the start_seq request.

token (input)
A fullword with the value of the token returned by (init_sub_dp) when the common runtime environment is initialized.

The token must identify a previously preinitialized environment for subroutines that are dormant at the time of the call.

Return codes
Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The environment was prepared for a sequence of calls.
4 The function code is not valid.
8 The indicated environment was already active; no action taken.
16 The token is not valid.
20 Sequence already started using token.

Usage notes
- (start_seq) can be used only in conjunction with a Language Environment environment initialized by (init_sub_dp) function code. A return code 4 is set for environments not initialized by (init_sub_dp).
- (start_seq) minimizes the overhead between calls by allowing Language Environment to perform as much activity as possible at the start of the sequence of calls.
- Only (call_sub) or (call_sub_addr) invocations are allowed between the (start_seq) and (end_seq) calls.
- The same token must be passed for all invocations of (call_sub) or (call_sub_addr) between the (start_seq) and (end_seq) function codes. You can vary the routine invoked.
- During a CEEPIPI call sequence, the user’s CEEPIPI driver must insure that the Language Environment recovery routines are never invoked when a program check or abend occurs in the user application code. One way to do this is to run with Trap (ON,NOSPIE), and also establish an ESTAE to handle errors when Language Environment is not active.

(term) — terminate environment

This invocation of CEEPIPI terminates the environment identified by the value given in token. This service is used for terminating environments created for subroutines or main routines.

CALL CEEPIPI (term, token, env_return_code)

term (input)
A fullword function code (integer value = 5) containing the termination request.
**Preinitialization services**

*token (input)*
A fullword with the value of the token of the environment to be terminated. This token is returned by a (init_main), (init_main_dp), (init_sub), or (init_sub_dp) 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 (call_sub) or (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 (call_sub) or (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 (call_sub) or (call_sub_addr).

**Return codes**
Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0  The environment was activated and termination was requested.

4  Non-valid function code.

8  If *token* was initialized by (init_main) or (init_sub), CEEPIPI(term) was called from a Language Environment-conforming routine.

If *token* was initialized by (init_main_dp), CEEPIPI(term) was called from a Language Environment-conforming routine that is not running in a (main_dp) environment, or *token* is already in use for another call to CEEPIPI

16  The *token* is not valid.

**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.

*(add_entry) — add an entry to the PreInit table*
This invocation of CEEPIPI 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.

**CALL**  
**CEEPIPI**  
**add_entry**  
**token**  
**routine_name**  
**routine_entry**  
**ceeexpbl_index**

*add_entry (input)*
A fullword function code (integer value = 6) containing the add_entry request.

*token (input)*
A fullword with the value of the token associated with the environment that adds this new routine. This token is returned by a (init_main), (init_main_dp), (init_sub), or (init_sub_dp) 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*.

The high-order bit of the entry_point address must be set to indicate the addressing mode for the routine. If the high-order bit is OFF, the routine is called in 24 bit addressing mode and the address must be a valid 24 bit address. If the high-order bit is ON, the routine is called in 31 bit addressing mode and the address must be a valid 31 bit address.

**ceexptbl_index** *(output)*
The 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.

**Note:** The environment that was preinitialized can be an XPLINK environment or a non-XPLINK environment. If the routine being added is an XPLINK routine, then the previously initialized environment must also be XPLINK.

**Return codes**
Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0  The routine was added to the PreInit table.

4  Non-valid function code.

8  If *token* was initialized by (init_main) or (init_sub), CEEPIPI(add_entry) was called from a Language Environment-conforming routine.

If *token* was initialized by (init_main_dp), CEEPIPI(add_entry) was called from a Language Environment-conforming routine that is not running in a (main_dp) environment, or *token* is already in use for another call to CEEPIPI

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.

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.

32  An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

38  Non-valid entry: A non-XPLINK subenvironment was preinitialized and the program that was being added is an XPLINK program.
Non-valid entry: The `routine_entry` had the high-order bit off indicating this routine is a 24 bit addressing mode routine but the environment is an XPLINK 31-bit environment. This is not valid.

**Usage notes**

- The PreInit table is built using the macros described in this topic. Therefore, its size is under the control of your application, not Language Environment.
- None of the routines in the PreInit table can be nested routines. All routines must be external routines.
- Language Environment uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the `routine_entry` is zero, and the `routine_name` is supplied, Language Environment uses the AMODE returned by the system loader. If the `routine_entry` is supplied, you must provide the AMODE in the high-order bit of the address.
- An `add_entry` of an XPLINK program into a non-XPLINK preinitialized sub-environment will be not valid. If the environment is non-XPLINK, then the subprogram added with the `add_entry` function must also be non-XPLINK. However, you can do an `add_entry` of a main XPLINK program into a non-XPLINK environment. When a `call_main` is done with this scenario the environment will switch to XPLINK in order to allow the program to run.

**(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.

```
CALL CEEPIPI(delete_entry, token, ceexptbl_index)
```

**delete_entry** (input)

Fullword function code (integer value = 11) containing the delete_entry request

**token** (input)

A fullword with the value of the token of the environment. This is the token returned by a (init_main), (init_main_dp), (init_sub), or (init_sub_dp) request.

**ceexptbl_index** (input)

The index into the PreInit table of the entry to delete.

**Return codes**

Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

- 0: The routine was deleted from the PreInit table
- 4: The function code is not valid.
- 8: If `token` was initialized by (init_main) or (init_sub), CEEPIPI(delete_entry) was called from an active environment.
  - If `token` was initialized by (init_main_dp), CEEPIPI(add_entry) was called from an active environment other than a (main_dp) environment, or `token` is already in use for another call to CEEPIPI.
  - No entries were deleted from the PreInit table.
- 16: The `token` is not valid
- 20: The PreInit table entry indicated by `ceexptbl_index` was empty.
The index passed is outside the range of the table.

The system request to delete the routine failed; the routine was not deleted from the PreInit table.

**Usage notes**

- The *token* must identify a previously preinitialized environment that is dormant at the time of the call.
- If the routine indicated by *ceexptbl_index* had been loaded by CEEPIPI, it will be deleted.
- *(delete_entry)* no longer issues return code 12 (the environment indicated by *token* was not created with a *(init_main)* request; the routine was not deleted from the PreInit table).

*(identify_entry) — identify an entry in the PreInit table*

This invocation of CEEPIPI identifies the language of the entry point for a routine in the PreInit table.

**CALL**

**CEEPIPI**

**identify_entry**

**token**

**ceexptbl_index**

**programming language**

**identify_entry** *(input)*

A fullword containing the *identify_entry* function code (integer value=13).

**token** *(input)*

A fullword with the value of the token of the environment. This is the token returned by a *(init_main)*, *(init_main_dp)*, *(init_sub)* or *(init_sub_dp)* request.

**ceexptbl_index** *(input)*

A fullword 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++
- 5  COBOL
- 10  PL/I
- 11  Enterprise PL/I for z/OS
- 15  Language Environment-enabled assembler
- 16  PL/X

**Return codes**

Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

- 0  The programming language has been returned.
- 4  Non-valid function code.
- 8  CEEPIPI was called from an active environment.
The token is not valid.

The PreInit table entry indicated by `ceexptbl_index` was empty.

The index passed is outside the range of the table.

**Usage notes**

- The token must identify a previously preinitialized environment that is dormant at the time of the call and was established with the `(init_main)`, `(init_main_dp)`, `(init_sub)` or `(init_sub_dp)` request.
- The `programming language` can be used by the driver to determine the format of the parameter list for the routine in cases where the language of the entry is not known.
- When a PreInit table entry contains multiple languages, `programming language` is the language of the entry point for the entry.

**((identify_environment) — identify the environment in the PreInit table)**

This invocation of CEEPIPI identifies the environment that was preinitialized.

**CALL CEEPIPI (identify_environment, token, pipi_environment)**

**identify_environment (input)**

A fullword containing the `identify_environment` function code (integer value=15).

**token (input)**

A fullword with the value of the token of the environment. This is the token returned by a `(init_main)`, `(init_main_dp)`, `(init_sub)` or `(init_sub_dp)` request.

**pipi_environment (output)**

A fullword (32 Bit) mask value will be returned. For information about the mask value, see Table 76 on page 456.

<table>
<thead>
<tr>
<th>pipi_environment</th>
<th>Mask value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ceepipi_main</code></td>
<td>X'80000000'</td>
<td>PreInit main environment is initialized.</td>
</tr>
<tr>
<td><code>ceepipi_enclave_initialized</code></td>
<td>X'40000000'</td>
<td>PreInit enclave is initialized.</td>
</tr>
<tr>
<td><code>ceepipi_dp_environment</code></td>
<td>X'20000000'</td>
<td>PreInit sub dp environment is initialized.</td>
</tr>
<tr>
<td><code>ceepipi_dp_seq_of_calls_active</code></td>
<td>X'10000000'</td>
<td>PreInit seq call function is active.</td>
</tr>
<tr>
<td><code>ceepipi_dp_exits_established</code></td>
<td>X'08000000'</td>
<td>PreInit sub dp exits is set.</td>
</tr>
<tr>
<td><code>ceepipi_sir_unregistered</code></td>
<td>X'04000000'</td>
<td>PreInit sir is registered.</td>
</tr>
<tr>
<td><code>ceepipi_sub_environment</code></td>
<td>X'02000000'</td>
<td>PreInit sub environment is initialized.</td>
</tr>
<tr>
<td><code>ceepipi_XPLINK_environment</code></td>
<td>X'01000000'</td>
<td>PreInit XPLINK environment is initialized.</td>
</tr>
<tr>
<td><code>ceepipi_init_main_dp_environment</code></td>
<td>X'00200000'</td>
<td>PreInit main dp environment is initialized.</td>
</tr>
</tbody>
</table>

**Note:** Mask bits other than those listed in the table may be nonzero. The meaning of these bits is not defined.

**Return codes**

Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:
The Preinitialization environment mask has been returned.

Non-valid function code.

CEEPIPI was called from an active environment.

The `token` is not valid.

**(identify_attributes) — identify the program attributes in the PreInit table**

This invocation of CEEPIPI identifies the program attributes of a program in the PreInit table.

```
CALL CEEPIPI (identify_attributes, token, ceexptbl_index(input))
```

**identify_attributes** *(input)*

A fullword containing the `identify_attributes` function code (integer value=16).

**token** *(input)*

A fullword with the value of the token of the environment. This is the token returned by a (init_main), (init_main_dp), (init_sub) or (init_sub_dp) request.

**ceexptbl_index** *(input)*

A fullword 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>
<thead>
<tr>
<th>program_attribute</th>
<th>Mask value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>loaded_by_pipi</td>
<td>X'80000000'</td>
<td>The Preinitialization entry was loaded by Language Environment</td>
</tr>
<tr>
<td>XPLINK_program</td>
<td>X'40000000'</td>
<td>The Preinitialization entry loaded is an XPLINK program</td>
</tr>
<tr>
<td>Address_not_resolved</td>
<td>X'20000000'</td>
<td>The Preinitialization entry could not be loaded</td>
</tr>
</tbody>
</table>

**Return codes**

Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0  The Preinitialization environment mask has been returned.

4  Non-valid function code.

8  CEEPIPI was called from an active environment.

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.
(set_user_word) -- set value to be used to initialize CAA user word

\[\text{\texttt{CALL CEEPIPI (set_user_word, token, value)}}\]

**set_user_word (input)**
A fullword containing the set_user_word function code (integer value = 17).

**token (input)**
A fullword with the value of the token of the environment. This is the token returned by a (init_main), (init_main_dp), (init_sub) or (init_sub_dp) request.

**value (input)**
A fullword value that will be used to initialize the user word in the initial thread CAA when the application is invoked using the (call_main), (call_sub), (call_sub_addr), (call_sub_addr_nochk), or (call_sub_addr_nochk2) functions for the passed-in environment token.

**Return codes**
Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

\[0\] The User Word has been set.
\[4\] Non-valid function code.
\[16\] The token is not valid.

**Usage notes**
- This value is saved in an area associated with the passed-in environment token. It is copied into the CAA for the initial thread when the next (call_main), (call_sub), (call_sub_addr), (call_sub_addr_nochk), or (call_sub_addr_nochk2) function is done to start an application. The application can then examine or update this user word in the CAA (CEECAA_USER_WORD). When the application ends, the final value in CEECAA_USER_WORD is not copied back into the area associated with the environment token. When the next application is started using a function such as (call_main), (call_sub), or (call_sub_addr), the user word value last established by (set_user_word) is used again.
- The user word associated with the environment token is initialized to 0 when (init_main), (init_main_dp), (init_sub), or (init_sub_dp) is done. The CAA for the initial process thread is initialized with 0 if no (set_user_word) function call has been done before the application is started.
- The user word in all CAAs other than the initial thread CAA is set to 0. The user word in all CAAs in nested enclaves is set to 0.
- When fork() is done, the user word in the CAA for the new process inherits the value that is in the CAA at the time fork() is done.
- The use of the CAA user word is not supported in the assembler user exit routine (CEEBXITA and related modules), or in the CEEPIPI service routines specified in the service routine vector (@LOAD, @DELETE, @GETSTORE, @FREESTORE, @EXCEPRTN, @MSGRTN).
- Any user code that runs on a CEEPIPI environment before the first (call_main), (call_sub), (call_sub_addr), (call_sub_addr_nochk), or (call_sub_addr_nochk2) request will see zero in the CAA_USER_WORD. Examples of this code include static constructors run for programs that get loaded when a CEEPIPI environment is initialized. Any changes to the CAA_USER_WORD made by this code are overlaid when the next (call_main), (call_sub), (call_sub_addr), (call_sub_addr_nochk), or (call_sub_addr_nochk2) is done for that environment.

(get_user_word) -- get value to be used to initialize CAA user word

\[\text{\texttt{CALL CEEPIPI (get_user_word, token, value)}}\]
get_user_word (input)
A fullword containing the get_user_word function code (integer value = 18).

token (input)
A fullword with the value of the token of the environment. This is the token returned by a (init_main),
(init_main_dp), (init_sub) or (init_sub_dp) request.

value (output)
A fullword that will be returned containing the current value that will be used to initialize the CAA user
word when the next application is invoked using the (call_main), (call_sub), (call_sub_addr),
(call_sub_addr_nochk), or (call_sub_addr_nochk2) functions.

Return codes
Register 15 contains a return code indicating the success or failure of the request. Possible return codes
(in decimal) are:

0  The current value of the User Word has been returned.
4  Non-valid function code.
16  The token is not valid.

Usage notes
• The value returned will be the one previously set by the last (set_user_word) request for this token. If no
(set_user_word) has yet been done for this token, 0 will be returned.

Service routines
Under Language Environment, you can specify several service routines to execute 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 108 on page 459.

<table>
<thead>
<tr>
<th>SERV_RTNS</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Word</td>
<td></td>
</tr>
<tr>
<td>@WorkArea</td>
<td></td>
</tr>
<tr>
<td>@LOAD</td>
<td></td>
</tr>
<tr>
<td>@DELETE</td>
<td></td>
</tr>
<tr>
<td>@GETSTORE</td>
<td></td>
</tr>
<tr>
<td>@FREESTORE</td>
<td></td>
</tr>
<tr>
<td>@EXCEPRTN</td>
<td></td>
</tr>
<tr>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>@MSGRTN</td>
<td></td>
</tr>
</tbody>
</table>

Figure 108: Format of service routine vector
The service routine vector is composed of a list of fullword addresses of routines that are used instead of Language Environment service routines. The list of addresses is preceded by the number of the addresses in the list, as specified in the *count* field of the vector. The *service_rtns* parameter that you specify in calls to *(init_main)* and *(init_sub)* contains the address of the vector itself. If this pointer is specified as zero (0), Language Environment routines are used instead of the service routines shown in Figure 108 on page 459.

The @GETSTORE and @FREESTORE service routines must be specified together; if one is zero, the other is automatically ignored. 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 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.

The service routines may be AMODE(31) / RMODE(ANY) if the application has no AMODE(24) programs. Otherwise the service routines must be AMODE(ANY) / RMODE(24).

**Count**

A fullword binary number representing the number of fullwords that follow. The *count* does not include itself. In Figure 108 on page 459, the count is 9. For each vector slot, a zero represents the absence of the routine, a nonzero represents the presence of a routine.

**User Word**

A fullword that is passed to the service routines. The *user word* is provided as a means for your routine to communicate to the service routines.

**@WorkArea**

An address of a work area of at least 256 bytes that is doubleword aligned. The first word of the area contains the length of the area provided. This parameter is required if service routines are present in the service routine vector. This length field must be initialized each time you bring up a new PreInit environment.

**@LOAD**

This routine loads named routines for application management. The parameter that is passed contains the following:

**Name_addr**

The fullword address of the name of the module to load (input parameter).

**Name_length**

A fixed binary(31) length of the module name (input parameter).

**User_word**

A fullword user field (input parameter).

**Load_point**

Either zero (0), or the address where the @LOAD routine is to store the load point address of the loaded routine (input and output parameter).

**Entry_point**

The fullword 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 Table 78 on page 461.
**Table 78: Return and reason codes from load (output)**

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Successful</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>Successful; loaded using SVC8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Unsuccessful; module loaded above the line when in AMODE(24)</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>Unsuccessful; load failed</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Unsuccessful; uncorrectable error occurred</td>
</tr>
</tbody>
</table>

@DELETE
This routine deletes routines for application management. The parameter that is passed contains the following:

**Name_addr**
The fullword 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 fullword user field (input parameter).

**Rsvd_word**
A fullword reserved for future use (input parameter); must be zero.

**Return code**
The return code from delete service (output).

**Reason code**
The reason code from delete service (output). The return and reason codes are listed in Table 79 on page 461.

**Table 79: Return and reason codes from delete service (output)**

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Successful</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>Unsuccessful; delete failed</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Unsuccessful; uncorrectable error occurred</td>
</tr>
</tbody>
</table>

@GETSTORE
This routine allocates storage on behalf of the storage manager. This routine can rely on the caller to provide a save area, which can be the @Workarea. The parameter list that is passed contains the following:

**Amount**
A fixed binary(31) amount of storage requested (input parameter).

**Subpool_no**
A fixed binary(31) subpool number 0-127 (input parameter). Language Environment allocates storage from the process-level storage pools.

**User word**
A fullword user field (input parameter).

**Flags**
A fullword flag area (input parameter), as shown in the following table. The remaining flag bits are reserved for future use and must be zero.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>ON</td>
<td>The storage that is required must be allocated below the 16 MB line.</td>
</tr>
</tbody>
</table>
Bit | Setting | Description
--- | --- | ---
OFF | OFF | The storage required can be allocated anywhere.
One | ON | The storage required was requested to be backed by 1 MB pages. This setting might be ignored.
OFF | OFF | The storage required was requested to be backed by the default 4 KB pages.

**Stg_address**
The fullword address of the storage that is obtained or zero (output parameter).

**Obtained**
A fixed binary(31) number of bytes obtained (output parameter).

**Return code**
The return code from @GETSTORE service (output parameter).

**Reason code**
The reason code from the @GETSTORE service (output parameter).

The return and reason codes are listed in Table 80 on page 462.

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Successful</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>Unsuccessful; uncorrectable error occurred</td>
</tr>
</tbody>
</table>

**@FREESTORE**
This routine frees storage on behalf of the storage manager. The parameter list passed contains the following:

**Amount**
The fixed binary(31) amount of storage to free (input parameter).

**Subpool_no**
The fixed binary(31) subpool number 0-127 (input parameter). Language Environment allocates storage from the process-level storage pools.

**User word**
A fullword user field (input parameter).

**Stg_address**
The fullword address of the storage to free (input parameter).

**Return code**
The return code from the @FREESTORE service (output).

**Reason code**
The reason code from the @FREESTORE service (output).

The return and reason codes are listed in Table 81 on page 462.

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Successful</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>Unsuccessful; uncorrectable error occurred</td>
</tr>
</tbody>
</table>

**@EXCEPRTN**
This routine traps program interruptions and abends for condition management. The parameter list passed contains the following:
**Handler_addr**
During an initialization call, this parameter contains the address of the Language Environment condition handler. During a termination call, this parameter contains a pointer to a fullword field containing zeroes.

**Environment_token**
A fullword Recovery Environment token (input). This token is different from the Preinitialization environment token used with CEEPIPI calls.

**User_word**
A fullword user field (input parameter)

**Abend_flags**
A fullword flag area containing abend flags (input)

**Check_flags**
A fullword flag area containing program check flags (input)

**Return code**
The return code from the @EXCEPRTN service (output).

**Reason code**
The reason code from the @EXCEPRTN service (output).

The exception router is responsible for trapping and routing exceptions. These are the services typically obtained via the ESTAE and ESPIE macros.

During initialization, Language Environment puts the address of the Language Environment condition handler in the first field of the above parameter list, and sets the environment token field to a value that must be passed on to the Language Environment condition handler. It also sets abend and check flags as appropriate, and then calls your exception router to establish an exception handler.

The meaning of the bits in the abend flags are given by the following declare:

```language
DCL
  1 abendflags, 2 system,
    3 abends bit(1), /* control for system abends desired */
    3 rsrv1 bit(15), /* reserved */
  2 user,
    3 abends bit(1), /* control for user abends desired */
    3 rsrv2 bit(15); /* reserved */
```

The meaning of the bits in the check flags is given by the following declare:

```language
DCL
  1 checkflags, 2 type,
    3 reserved3 bit(1),
    3 operation bit(1),
    3 privileged_operation bit(1),
    3 execute bit(1),
    3 protection bit(1),
    3 addressing bit(1),
    3 specification bit(1),
    3 data bit(1),
    3 fixed_overflow bit(1),
    3 fixed_divide bit(1),
    3 decimal_overflow bit(1),
    3 decimal_divide bit(1),
    3 exponent_overflow bit(1),
    3 exponent_underflow bit(1),
    3 significance bit(1),
    3 float_divide bit(1),
  2 reserved4 bit(16);
```

The return and reason codes that the exception router must use are listed in Table 82 on page 463.

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Successful</td>
</tr>
</tbody>
</table>
Table 82: Return and reason codes for the exception router (continued)

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>Unsuccessful; the exit could not be established or removed</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Unsuccessful; unrecoverable error occurred</td>
</tr>
</tbody>
</table>

When an exception occurs, the exception handler must determine if the Language Environment condition handler is interested in the exception (by examining abend and check flags). If the condition handler is not interested in the exception, the exception handler must treat the program as in error, but can assume the environment for the thread to be functional and reusable. If the condition handler is interested in the exception, the exception handler must invoke the condition handler, passing the parameters listed in Table 83 on page 464.

Table 83: Parameters for Language Environment condition handler

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Attributes</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment Token</td>
<td>Fixed Bin(31)</td>
<td>Input</td>
</tr>
<tr>
<td>Address of SDWA</td>
<td>Pointer</td>
<td>Input</td>
</tr>
<tr>
<td>Return Code</td>
<td>Fixed Bin(31)</td>
<td>Output</td>
</tr>
<tr>
<td>Reason Code</td>
<td>Fixed Bin(31)</td>
<td>Output</td>
</tr>
</tbody>
</table>

The return and reason codes upon return from the Language Environment condition handler are listed in Table 84 on page 464.

Table 84: Return and reason codes from the Language Environment condition handler

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Continue with the exception. Percolate the exception taking whatever action would have been taken had it not been handled at all. In this case, your exception handler can assume the environment for the thread to be functional and reusable.</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>Continue with the exception. Percolate the exception taking whatever action would have been taken had it not been handled at all. In this case, the environment for the thread is probably unreliable and not reusable. A forced termination is suggested.</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>Resume execution using the updated SDWA. The invoked Language Environment condition handler will have already used the SETRP RTM macro to set the SDWA for correct resumption.</td>
</tr>
</tbody>
</table>

During termination, the exception router is invoked with the condition handler address (first parameter) set to zero to de-establish the exit (if it was established during initialization).

When a nested enclave is created, Language Environment calls the exception router to establish another exception handler exit, and then makes a call to de-establish it when the nested enclave terminates. If an exception occurs while the second exit is active, special processing is performed. Depending on what this second exception is, either the first exception will not be retried, or processing will continue on the first exception by requesting retry for the second exception.

If the Language Environment condition handler determines that execution should resume for an exception, it will set the SDWA with SETRP and return with return/reason codes 4/0. Execution will resume in library code or in user code, depending on what the exception was.

The exception handler must be capable of restoring all the registers from the SDWA when control is given to the retry routine. The ESPIE and ESTAE services are capable of accomplishing this.
In using the exception router service:

- The exception handler should not invoke the Language Environment condition handler if active I/O has been halted and is not restorable.
- This service requires an XA or ESA environment.

If an exception occurs while the exception handler is in control before another exception handler exit has been stacked, the exception handler should assume that the exception could not be handled and that the environment for the program (thread) is damaged. In this case, the exception handler should force termination of the preinitialized environment.

When @EXCEPRTN is specified, the following items are not supported:

- XPLINK applications
- POSIX(ON) applications
- DYNDUMP settings other than DYNDUMP(NODYNAMIC)
- IMS applications
- Applications that use Binary Floating Point (BFP) or Decimal Floating Point (DFP) numbers
- Applications that use the Compare-and-Trap family of instructions

Notes:

1. If the passed-in SDWA from the exception handler to the Language Environment condition handler does not contain valid high registers, the "HR_VALID" flag bit in the Machine State "FLAGS" field will be off, indicating that the saved high registers are not valid.
2. If a nested enclave ends because of an unhandled condition and a 4094-40 ABEND is declared, the high registers may not be valid in the Machine State that contains information about the 4094-40 ABEND.
3. If registers in the passed-in SDWA at the time of interrupt (in the SDWAGRVS field) are not appropriate or recognizable, and Language Environment instead saves the registers from the SDWASRVS field in the Machine State, the high registers may not be valid in the Machine State.

@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, messages can be 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 fullword user field (input parameter).

**Line_length**

The fixed binary(31) size of the output line length. This is used when Message is zero (output parameter).

**Return and reason codes**

Two fullwords containing the return and reason codes listed in Table 85 on page 465 (output parameters).

<table>
<thead>
<tr>
<th>Return code</th>
<th>Reason code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Successful</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Unsuccessful; uncorrectable error occurred</td>
</tr>
</tbody>
</table>
An example program invocation of CEEPIPI

This section includes a sample of a PreInit assembler driver program. This assembler program called ASMPIPI invokes CEEPIPI to:

- Initialize a subroutine environment under Language Environment
- Load and call a reentrant HLL subroutine
- Terminate the Language Environment environment

Following the assembler program are examples of the program HLLPIPI written in C, COBOL, and PL/I. HLLPIPI is called by an assembler program, ASMPIPI. ASMPIPI uses the Language Environment preinitialized program subroutine call interface. You can use the assembler program to call the HLL versions of HLLPIPI.

```assembly
* COMPILATION UNIT: LEASMPIPI
***********************************************************************
* Function : CEEPIPI - Initialize the Preinit                      *
*                         environment, call a Preinit                 *
*                         HLL program, and terminate the environment. *
* 1. Call CEEPIPI to initialize a subroutine environment.              *
* 2. Call CEEPIPI to load and call a reentrant HLL subroutine.         *
* 3. Call CEEPIPI to terminate the Preinit environment.               *
* Note: ASMPIPI is not reentrant.                                     *
***********************************************************************
* =====================================================================
* Standard program entry conventions.                                 *
***********************************************************************
ASMPIPI CSECT
STM   R14,R12,12(R13)    Save caller's registers
LR    R12,R15            Get base address
USING ASMPIPI,R12        Identify base register
ST    R13,SAVE+4         Back-chain the save area
LA    R15,SAVE           Get addr of this routine's save area
ST    R15,8(R13)         Forward-chain in caller's save area
LR    R13,R15            R13 -> save area of this routine

* Load CEEPIPI service routine into main storage.
* LOAD  EP=CEEPIPI         Load CEEPIPI routine dynamically
ST    R0,PPRTNPTR        Save the addr of CEEPIPI routine

* Initialize a Preinit subroutine environment.
* INIT_ENV EQU *
LA    R9,PPRTBL          Get address of Preinit Table
LA    R5,CEXPTBL         Cexptbl-addr -> Preinit Table
L     R15,PPRTNTPTR      Get addr of CEEPIPI routine
CALL  (15),(INIT_SUB,@CEXPTBL,@SRVRTNS,RUNITMOPT,TOKEN) * Invoke CEEPIPI routine
* Check return code:
LTR   R2,R15               Is R15 = zero?
BZ    CSUB                 Yes (success).. go to next section
* No (failure) .. issue message
WTO   'ASMPIPI : call to (INIT_SUB) failed',ROUTCDE=11
C     R2,=F'8'           Check for partial initialization
BE    TSUB                 Yes.. go Preinit termination
* WTO 'ASMPIPI : INIT_SUB failure RC is not 8.',ROUTCDE=11
ABEND (R2),DUMP          Abend with bad RC and dump memory
* Call the subroutine, which is loaded by LE
* CSUB    EQU *
LA    R15,PPRTNTPTR      Get address of CEEPIPI routine
CALL  (15),(CALL_SUB,PTBINDEX,TOKEN,PARMPTR,                  X
SUBRETC,SUBRSNC,SUBFBC)    Invoke CEEPIPI routine
* Check return code:
```
Using preinitialization services

467
HLLPIPI examples

Following is an example of a C subroutine called by ASMPIPI:

```c
/*Module/File Name:  EDCPIPI  */
/*------------------------------*/
/* HLLPIPI is called by an assembler program, ASMPIPI. */
/* ASMPIPI uses the LE preinitialized program */
/* subroutine call interface. HLLPIPI can be written */
/* in COBOL, C, or PL/I. */
/*------------------------------*/
#include <stdio.h>
#include <string.h>
#include <time.h>
#pragma linkage(HLLPIPI, fetchable)
HLLPIPI ()
{
    printf ( "C subroutine beginning\n" );
    printf ( "Called using LE PreInit call\n" );
    printf ( "Subroutine interface.\n" );
    printf ( "C subroutine returns to caller\n" );
}
```

Following is an example of a COBOL program called by ASMPIPI:

```cobol
CBL LIB,QUOTE
   *Module/File Name: IGZTPPIPI
   *-----------------------------------------------------------------------
   * HLLPIPI is called by an assembler program, ASMPIPI. *
   * ASMPIPI uses the LE preinitialized program *
   * subroutine call interface. HLLPIPI can be written *
   * in COBOL, C, or PL/I. *
   *-----------------------------------------------------------------------
IDENTIFICATION DIVISION.
PROGRAM-ID. HLLPIPI.
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
   DISPLAY "COBOL subprogram beginning".
   DISPLAY "Called using LE Preinitialization ".
   DISPLAY "Call subroutine interface.".
   DISPLAY "COBOL subprogram returns to caller.".
   GOBACK.
```

Following is an example of a routine called by ASMPIPI:

```pli
/*Module/File Name:  IBMPIPI                              */
/*------------------------------*/
/* HLLPIPI is called by an assembler program, ASMPIPI. */
/* ASMPIPI uses the LE preinitialized program */
/* subroutine call interface. HLLPIPI can be written */
/* in COBOL, C, or PL/I. */
/*------------------------------*/
HLLPIPI: PROC OPTIONS(FETCHABLE);
   DCL RESULT FIXED BIN(31,0) INIT(0);
   PUT SKIP LIST ('HLLPIPI  : PLI subroutine beginning.');
   PUT SKIP LIST ('HLLPIPI  : Called LE PIPI Call ');
   PUT SKIP LIST ('HLLPIPI  : Subroutine interface. ');
   PUT SKIP LIST ('HLLPIPI  : PLI program returns to caller.');
   RETURN;
END HLLPIPI;
```
Chapter 31. Using nested enclaves

An enclave is a logical runtime structure that supports the execution of a collection of routines (see Chapter 13, “Program management model,” on page 139 for a detailed description of Language Environment enclaves).

Language Environment explicitly supports the execution of a single enclave within a Language Environment process. However, by using the system services and language constructs described in this topic, you can create an additional, or nested, enclave and initiate its execution within the same process.

The enclave that issues a call to system services or language constructs to create a nested enclave is called the parent enclave. The nested enclave that is created is called the child enclave. The child must be a main routine; a link to a subroutine by commands and language constructs is not supported under Language Environment.

If a process contains nested enclaves, none or only one enclave can be running with POSIX(ON).

Creating child enclaves

In Language Environment, you can use the following methods to create a child enclave:

- Under CICS, the EXEC CICS LINK and EXEC CICS XCTL commands. For more information about these commands, go to CICS Transaction Server for z/OS (www.ibm.com/support/knowledgecenter/SSGMGV/welcome.html).
- Under z/OS, the SVC LINK macro
- Under z/OS, the C system() function (see z/OS XL C/C++ Programming Guide for more information about system()).
- Under z/OS, the PL/I FETCH and CALL to any of the following PL/I routines with PROC OPTIONS(MAIN) specified:
  - Enterprise PL/I for z/OS
  - PL/I for MVS & VM
  - OS PL/I Version 2
  - OS PL/I Version 1 Release 5.1
  - Relinked OS PL/I Version 1 Release 3.0 – 5.1

Such a routine, called a fetchable main in this section, can only be introduced by a FETCH and CALL from a PL/I routine. COBOL cannot dynamically call a PL/I main and C cannot issue a fetch() against a PL/I main. In addition, a fetchable main cannot be dynamically loaded using the CEELOAD macro.

The routine performing the FETCH and CALL must be compiled with the Enterprise PL/I for z/OS or the PL/I for MVS & VM compiler, or be a relinked OS PL/I routine.

If the target routine of any of these commands is not written in a Language Environment-conforming HLL or Language Environment-conforming assembler, no nested enclave is created.

XPLINK considerations

A nested enclave situation where the parent enclave is running in an XPLINK(OFF) environment and the child enclave requires XPLINK(ON) is not supported. A parent enclave running XPLINK(ON) will support a nested child enclave of either XPLINK(ON) or XPLINK(OFF). In the latter case, the application in the child enclave will go through compatibility glue code when calling the C RTL (that is, the child enclave will run with an environment with the XPLINK runtime option forced ON).
**COBOL considerations**

In a non-CICS environment, OS/VS COBOL programs are supported in a single enclave only.

**PL/I considerations**

PL/I MTF is supported in the initial enclave only. If PL/I MTF is found in a nested enclave, Language Environment diagnoses it as an error. If a PL/I MTF application contains nested enclaves, the initial enclave must contain a single task. Violation of this rule is not diagnosed and is likely to cause unpredictable results.

**Determining the behavior of child enclaves**

If you want to create a child enclave, you need to consider the following factors:

- The language of the main routine in the child enclave
- The sources from which each type of child enclave gets runtime options
- The default condition handling behavior of each type of child enclave
- The setting of the TRAP runtime option in the parent and the child enclave

All of these interrelated factors affect the behavior, particularly the condition handling, of the created enclave. The sections that follow describe how the child enclaves created by each method (EXEC CICS LINK, EXEC CICS XCTL, SVC LINK, C function, and PL/I FETCH and CALL of a fetchable main) will behave.

**Creating child enclaves with EXEC CICS LINK or EXEC CICS XCTL**

If your C, C++, COBOL, or PL/I application uses EXEC CICS commands, you must also link-edit the EXEC CICS interface stub, DFHELII, with your application. To be link-edited with your application, DFHELII must be available in the link-edit SYSLIB concatenation.

For more information about the EXEC CICS LINK and EXEC CICS XCTL commands, go to CICS Transaction Server for z/OS (www.ibm.com/support/knowledgecenter/SSGMGV/welcome.html).

**How runtime options affect child enclaves**

The child enclave gets its runtime options from one of the sources discussed in “Specifying runtime options under CICS” on page 354. The runtime options are completely independent of the creating enclave, and can be set on an enclave-by-enclave basis.

Some of the methods for setting runtime options might slow down your transaction. Follow these suggestions to improve performance:

- If you need to specify options in CEEUOPT specify only those options that are different from system defaults.
- Before putting transactions into production, request a storage report (using the RPTSTG runtime option) to minimize the number of GETMAINs and FREEMAINs required by the transactions.
- Ensure that VS COBOL II transactions are not link-edited with IGZETUN and IGZEOPT, which are no longer supported and which cause an informational message to be logged. Logging this message for every transaction inhibits system performance. The sample user condition handler CEEWUCHA can be used to prevent this informational message from being logged. See the Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733) for more information.

**How conditions arising in child enclaves are handled**

This section describes the default condition handling for child enclaves created by EXEC CICS LINK or EXEC CICS XCTL.
Condition handling varies depending on the source of the condition, and whether an EXEC CICS HANDLE ABEND is active:

- If a Language Environment or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs, the CICS thread is terminated. This occurs even if a CICS HANDLE ABEND is active, because CICS HANDLE ABEND does not gain control in the event of a Language Environment abend.

- If a software condition of severity 2 or greater occurs, Language Environment condition handling takes place. If the condition remains unhandled, the problem is not percolated to the parent enclave. The CICS thread is terminated with an abend. These actions take place even if a CICS HANDLE ABEND is active, because CICS HANDLE ABEND does not gain control in the event of a Language Environment software condition.

- If a user abend or program check occurs, the following actions take place:
  - If no EXEC CICS HANDLE ABEND is active, and TRAP(ON) is set in the child enclave, Language Environment condition handling takes place. If the abend or program check remains unhandled, the problem is not propagated to the parent enclave. The CICS thread is terminated with an abend.
  - An active EXEC CICS HANDLE ABEND overrides the setting of TRAP. The action defined by the EXEC CICS HANDLE ABEND takes place.

Creating child enclaves by calling a second main program

The behavior of a child enclave created by calling a second main program is determined by the language of its main or initializing routine: C, C++, COBOL, Fortran, PL/I, or Language Environment-conforming assembler (generated by use of the CEEENTRY and associated macros).

How runtime options affect child enclaves

Runtime options will be processed in the normal manner for enclaves created because of a call to a second main, that is, programmer defaults present in the load module will be merged, options in the command line equivalent will also be processed, as will options passed by the assembler user exit if present.

How conditions arising in child enclaves are handled

The command-line equivalent is determined in the same manner as for a SVC LINK.

Creating child enclaves using SVC LINK

The behavior of a child enclave created by an SVC LINK is determined by the language of its main routine: C, C++, COBOL, Fortran, PL/I, or Language Environment-conforming assembler (generated by use of the CEEENTRY and associated macros).

When issuing a LINK to a routine, the high-order bit must be set on for the last word of the parameter list. To do this, set VL=1 on the LINK assembler macro.

How runtime options affect child enclaves

Child enclaves created by an SVC LINK get runtime options differently, depending on the language that the main routine of the child enclave is written in.

Child enclave has a C, C++, Fortran, PL/I, or Language Environment-conforming assembler main routine

If the main routine of the child enclave is written in C, C++, Fortran, PL/I, or in Language Environment-conforming assembler, the child enclave gets its runtime options through a merge from the usual sources (see Chapter 9, “Using runtime options,” on page 101 for more information). Therefore, you can set runtime options on an enclave-by-enclave basis.
Using nested enclaves

**Child enclave has a COBOL main program**

If the main program of the child enclave is written in COBOL, the child enclave inherits the runtime options of the creating enclave. Therefore, you cannot set runtime options on an enclave-by-enclave basis.

**How conditions arising in child enclaves are handled**

If a Language Environment or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs in a child enclave created by SVC LINK, regardless of the language of its main, the entire process is terminated.

Condition handling in child enclaves created by SVC LINK varies, depending on the language of the child's main routine, the setting of the TRAP runtime option in the parent and child enclaves, and the type of condition. Refer to one of the following tables to see what happens when a condition remains unhandled in a child enclave.

<table>
<thead>
<tr>
<th>If the child enclave was created by:</th>
<th>See:</th>
</tr>
</thead>
<tbody>
<tr>
<td>An SVC LINK and has a C, C++, or Language Environment-conforming assembler main routine</td>
<td>Table 87 on page 472</td>
</tr>
<tr>
<td>An SVC LINK and has a COBOL main program</td>
<td>Table 88 on page 473</td>
</tr>
<tr>
<td>An SVC LINK and has a Fortran or PL/I main routine</td>
<td>Table 89 on page 473</td>
</tr>
</tbody>
</table>

You should always run your applications with TRAP(ON) or your results might be unpredictable.

**Child enclave has a C, C++, or Language Environment-conforming assembler main routine**

Table 87 on page 472 shows the unhandled condition behavior.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(OFF)</th>
<th>Parent enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unhandled condition severity 0 or 1</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
</tr>
<tr>
<td>Unhandled condition severity 2 or above</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Resume parent enclave, and ignore condition</td>
</tr>
<tr>
<td>Non-Language Environment abend</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with original abend code</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with original abend code</td>
</tr>
<tr>
<td>Program check</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with abend U4036, Reason Code=2</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with abend S0Cx</td>
</tr>
</tbody>
</table>
**Child enclave has a COBOL main program**

Child enclaves created by SVC LINK that have a COBOL main program inherit the runtime options of the parent enclave that created them. Therefore, the TRAP setting of the parent and child enclaves is always the same.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(OFF)</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(OFF)</td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(OFF)</td>
</tr>
<tr>
<td>Unhandled condition severity 0 or 1</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
</tr>
<tr>
<td>Unhandled condition severity 2 or above</td>
<td>Signal CEE391 (Severity=1, Message Number=3361) in parent enclave</td>
<td>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with abend U4094 RC=40</td>
<td></td>
</tr>
<tr>
<td>Non-Language Environment abend</td>
<td>Signal CEE391 in parent enclave</td>
<td>Process terminated with original abend code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program check</td>
<td>Signal CEE391 in parent enclave</td>
<td>Process terminated with abend S0Cx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Child enclave has a Fortran or PL/I main routine**

Table 89 on page 473 lists unhandled condition behavior.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(OFF)</th>
<th>Parent enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(OFF)</td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(OFF)</td>
</tr>
<tr>
<td>Unhandled condition severity 0 or 1</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
</tr>
<tr>
<td>Unhandled condition severity 2 or above</td>
<td>Signal CEE391 in parent enclave</td>
<td>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with abend U4094 RC=40</td>
<td></td>
</tr>
<tr>
<td>Non-Language Environment abend</td>
<td>Signal CEE391 in parent enclave</td>
<td>Process terminated with original abend code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program check</td>
<td>Signal CEE391 in parent enclave</td>
<td>Process terminated with abend S0Cx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Creating child enclaves using the C system() function

Child enclaves created by the C system() function get runtime options through a merge from the usual sources. See Chapter 9, “Using runtime options,” on page 101 for more information. Therefore, you can set runtime options on an enclave-by-enclave basis. See z/OS XL C/C++ Runtime Library Reference for information on the system() function when running with POSIX(ON).

When you perform a system() function to a COBOL program, in the form:

```
system("PGM=program_name,PARM='...'")
```

the runtime options specified in the PARM= portion of the system() function are ignored. However, runtime options are merged from CEEDOPT, CEEUOPT, and the CEEAUE_A_OPTIONS from the assembler user exit.

z/OS UNIX considerations

To create a nested enclave under z/OS UNIX, you must either:

- Be running with POSIX(OFF) and issue system(), or
- Be running with POSIX(ON) and have set the environment variables to signal that you want to establish a nested enclave. You can use the __POSIX_SYSTEM environment variable to cause a system() to establish a nested enclave instead of performing a spawn(). __POSIX_SYSTEM can be set to NO, No, or no.

The system() function is not thread safe. It cannot be called simultaneously from more than one thread. A multi-threaded application must ensure that no more than one system() call is ever outstanding from the various threads. If this restriction is violated, unpredictable results may occur. In a multiple enclave environment, the first enclave must be running with POSIX(ON) and all other nested enclaves must be running with POSIX(OFF).

How conditions arising in child enclaves are handled

If a Language Environment- or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs in a child enclave created by a call to system(), the entire process is terminated.

Depending on what the settings of the TRAP runtime option are in the parent and child enclave, the following might cause the child enclave to terminate:

- Unhandled user abend
- Unhandled program check

TRAP(ON | OFF) effects for enclaves created by system()

Table 90 on page 474 describes the effects of TRAP(ON|OFF) for enclaves that are created by the system() function.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(OFF)</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(OFF)</td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(OFF)</td>
</tr>
<tr>
<td>Unhandled condition severity 0 or 1</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
</tr>
<tr>
<td>Unhandled condition severity 2 or above</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Resume parent enclave, and ignore condition</td>
</tr>
</tbody>
</table>

Using nested enclaves
### Table 90: Unhandled condition behavior in a system()-created child enclave (continued)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(OFF)</th>
<th>Parent enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(OFF)</td>
<td>Child enclave TRAP(OFF)</td>
</tr>
<tr>
<td>Non-Language Environment abend</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with original abend code</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with original abend code</td>
</tr>
<tr>
<td>Program check</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with abend U4036, Reason Code=2</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with abend S0Cx</td>
</tr>
</tbody>
</table>

### Creating child enclaves containing a PL/I fetchable main

Fetch and call considerations of PL/I fetchable mains are discussed in “Special fetch and call considerations” on page 476.

#### How runtime options affect child enclaves

Child enclaves created when you issue a FETCH and CALL of a fetchable main get runtime options through a merge from the usual sources (see Chapter 9, “Using runtime options,” on page 101 for more information). Therefore, you can set runtime options on an enclave-by-enclave basis.

#### How conditions arising in child enclaves are handled

If a Language Environment or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs in a child enclave that contains a fetchable main, the entire process is terminated.

Depending on what the settings of the TRAP runtime option are in the parent and child enclave, the following might cause the child enclave to terminate:
- Unhandled user abend
- Unhandled program check

Table 91 on page 475 describes the unhandled condition behavior in a child enclave.

### Table 91: Unhandled condition behavior in a child enclave that contains a PL/I fetchable main

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(OFF)</th>
<th>Parent enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(OFF)</td>
<td>Child enclave TRAP(OFF)</td>
</tr>
<tr>
<td>Unhandled condition severity 0 or 1</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
<td>Resume child enclave</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unhandled condition severity 2 or above</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Resume parent enclave, and ignore condition</td>
</tr>
<tr>
<td>Non-Language Environment abend</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with original abend code</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with original abend code</td>
</tr>
</tbody>
</table>
**Table 91: Unhandled condition behavior in a child enclave that contains a PL/I fetchable main (continued)**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(ON)</th>
<th>Parent enclave TRAP(OFF)</th>
<th>Parent enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(ON)</td>
<td>Child enclave TRAP(OFF)</td>
<td>Child enclave TRAP(OFF)</td>
</tr>
<tr>
<td>Program check</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with abend U4036, Reason code=2</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with abend S0Cx</td>
</tr>
</tbody>
</table>

**Special fetch and call considerations**

You should not recursively fetch and call the fetchable main from within the child enclave; results are unpredictable if you do.

The load module that is the target of the FETCH and CALL is reentrant if all routines in the load module are reentrant. (See Chapter 11, “Making your application reentrant,” on page 121 for more information on reentrancy.)

Language Environment relies on the underlying operating system for the management of load module attributes. In general, multiple calls of the same load module are supported for load modules that are any of the following:

- **Reentrant**
  - It is recommended that your target load module be reentrant.

- **Nonreentrant but serially reusable**
  - You should ensure that the main procedure of a nonreentrant but serially reusable load module is self-initializing. Results are unpredictable otherwise.

- **Nonreentrant and non-serially reusable**
  - If a nonreentrant and non-serially reusable load module is called multiple times, each new call brings in a fresh copy of the load module. That is, there are two copies of the load module in storage: one from FETCH and one from CALL. Even though there are two copies of the load module in storage, you need only one PL/I RELEASE statement because upon return from the created enclave the load module loaded by CALL is deleted by the operating system. You need only release the load module loaded by FETCH.

**Other nested enclave considerations**

The following sections contain other information you might need to know when creating nested enclaves. The topics include:

- The string that CEE3PRM and CEE3PR2 return for each type of child enclave (see z/OS Language Environment Programming Reference for more information about the CEE3PRM and CEE3PR2 callable service)
- The return and reason codes that are returned on termination of the child enclave
- How the assembler user exit handles nested enclaves
- Whether the message file is closed on return from a child enclave
- z/OS UNIX considerations
- AMODE considerations
What the enclave returns from CEE3PRM and CEE3PR2

CEE3PRM and CEE3PR2 return to the calling routine the user parameter string that was specified at program invocation. Only program arguments are returned.

See Table 92 on page 477 to determine whether a user parameter string was passed to your routine, and where the user parameter string is found. This depends on the method you used to create the child enclave, the language of the routine in the child enclave, and the PLIST, TARGET, or SYSTEM setting of the main routine in the child enclave. If a user parameter string was passed to your routine, the user parameter string is extracted from the command-line equivalent for your routine (shown in Table 93 on page 477) and returned to you.

**Note:** Under CICS, CEE3PRM and CEE3PR2 always return a blank string.

### Table 92: Determining the command-line equivalent.

<table>
<thead>
<tr>
<th>Language</th>
<th>Option</th>
<th>Suboption</th>
<th>system()</th>
<th>SVC LINK</th>
<th>FETCH/CALL of a PL/I main</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>#pragma runopts(PLIST)</td>
<td>HOST, MVS</td>
<td>PARM=, or the parameter string from the command string passed to system()</td>
<td>Halfword length-prefixed string pointed to by R1</td>
<td>Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CICS, IMS, OS, or TSO</td>
<td>Not available</td>
<td>Not available</td>
<td>Not allowed</td>
</tr>
<tr>
<td>C++</td>
<td>PLIST and TARGET compiler options</td>
<td>Default</td>
<td>PARM=, or the parameter string from the command string passed to system()</td>
<td>Halfword length-prefixed string pointed to by R1</td>
<td>Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PLIST(OS) or TARGET(IMS)</td>
<td>Not available</td>
<td>Not available</td>
<td>Not allowed</td>
</tr>
<tr>
<td>COBOL</td>
<td>N/A</td>
<td>N/A</td>
<td>Null</td>
<td>Not allowed</td>
<td></td>
</tr>
<tr>
<td>Fortran</td>
<td>N/A</td>
<td>Not available</td>
<td>Halfword length-prefixed string pointed to by R1</td>
<td>Not allowed</td>
<td></td>
</tr>
<tr>
<td>PL/I</td>
<td>SYSTEM compiler option</td>
<td>MVS</td>
<td>PARM=, or the parameter string from the command string passed to system()</td>
<td>Halfword length-prefixed string pointed to by R1</td>
<td>User parameters passed through CALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CICS, IMS, TSO</td>
<td>Not available</td>
<td>Not available</td>
<td>SYSTEM(CICS) not supported; others not available.</td>
</tr>
<tr>
<td>Language Environment-conforming assembler</td>
<td>CEENTRY PLIST=</td>
<td>HOST, MVS</td>
<td>PARM=, or the parameter string from the command string passed to system()</td>
<td>Halfword length-prefixed string pointed to by R1</td>
<td>Not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CICS, IMS, OS, or TSO</td>
<td>Not available</td>
<td>Not available</td>
<td>Not allowed</td>
</tr>
</tbody>
</table>

If Table 92 on page 477 indicates that a parameter string was passed to your routine at invocation, the string is extracted from the command-line equivalent listed in the right-hand column of Table 93 on page 477. The command-line equivalent depends on the language of your routine and the runtime options specified for it.

### Table 93: Determining the order of runtime options and program arguments.

<table>
<thead>
<tr>
<th>Language of routine</th>
<th>Runtime options in effect?</th>
<th>Order of runtime options and program arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>#pragma runopts(EXECOPS)</td>
<td>runtime options / user parms</td>
</tr>
<tr>
<td></td>
<td>#pragma runopts(NOEXECOPS)</td>
<td>entire string is user parms</td>
</tr>
</tbody>
</table>
Using nested enclaves

Table 93: Determining the order of runtime options and program arguments. (continued)

<table>
<thead>
<tr>
<th>Language of routine</th>
<th>Runtime options in effect?</th>
<th>Order of runtime options and program arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>Compiled with EXECOPS (default)</td>
<td>runtime options / user parms</td>
</tr>
<tr>
<td></td>
<td>Compiled with NOEXECOPS</td>
<td>entire string is user parms</td>
</tr>
<tr>
<td>COBOL</td>
<td>CBLOPTS(ON)</td>
<td>user parms / runtime options</td>
</tr>
<tr>
<td></td>
<td>CBLOPTS(OFF)</td>
<td>runtime options / user parms</td>
</tr>
<tr>
<td>Fortran</td>
<td></td>
<td>runtime options / user parms</td>
</tr>
<tr>
<td>PL/I</td>
<td>PROC OPTIONS(NOEXECOPS) or SYSTEM(CICS</td>
<td>IMS</td>
</tr>
<tr>
<td></td>
<td>PROC OPTIONS(NOEXECOPS) is specified, or NOEXECOPS is not specified but SYSTEM (CICS</td>
<td>IMS</td>
</tr>
</tbody>
</table>

Language Environment-conforming assembler

<table>
<thead>
<tr>
<th>Language Environment-conforming assembler</th>
<th>CEENTRY EXECOPS=ON</th>
<th>runtime options / user parms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CEENTRY EXECOPS=OFF</td>
<td>entire string is user parms</td>
</tr>
</tbody>
</table>

Finding the return and reason code from the enclave

The following list tells where to look for the return and reason codes that are returned to the parent enclave when a child enclaves terminates:

- EXEC CICS LINK or EXEC CICS XCTL

  If the CICS thread was not terminated, the return code is placed in the optional RESP2 field of EXEC CICS LINK or EXEC CICS XCTL. The reason code is discarded.

- SVC LINK to a child enclave with a main routine written in any Language Environment-conforming language

  If the process was not terminated, the return code is reported in R15. (See “Managing return codes in Language Environment” on page 132 for more information.) The reason code is discarded.

- C's system() function

  If the target command or program of system() cannot be started, “-1” is returned as the function value of system(). Otherwise, the return code of the child enclave is reported as the function value of system(), and the reason code is discarded. (See z/OS XL C/C++ Programming Guide for more information about the system() function.)

- FETCH and CALL of a fetchable main

  Normally, the enclave return code and reason code are discarded when control returns to a parent enclave from a child enclave. However, in the parent enclave, you can specify the OPTIONS(ASSEMBLER RETCODE) option of the entry constant for the main procedure of the child enclave. This causes the enclave return code of the child enclave to be saved in R15 as the PL/I return code. You can then interrogate that value by using the PLIRETV built-in function in the parent enclave.
Assembler user exit

An assembler user exit (CEEBXITA) is driven for enclave initialization and enclave termination regardless of whether the enclave is the first enclave created in the process or a nested enclave. The assembler user exit differentiates between first and nested enclave initialization.

Message file

The message file is not closed when control returns from a child enclave.

COBOL multithreading considerations

When COBOL is run in a multithread environment or a PL/I multitasking environment, a nested enclave cannot be created. An attempt to create a nested enclave results in a severity 3 condition being generated.

z/OS UNIX considerations

The following restrictions must be considered when running with POSIX(OFF) or POSIX(ON):

- In Language Environment a process can have only one enclave that is running with POSIX(ON), and that enclave must be the first enclave if that process contains multiple enclaves. All nested enclaves must be enclaves with POSIX(OFF).
- C exec() can only be issued from a single-thread enclave.

Any violations of the above restrictions result in a severity 3 condition being generated.

AMODE considerations

In a non-CICS environment ALL31 should have the same setting for all enclaves within a process. You cannot invoke a nested enclave that requires ALL31(OFF) from an enclave running with ALL31(ON).
Appendix A. Prelinking an application

This topic describes how to prelink your programs under Language Environment. Unless otherwise indicated, the prelinking process applies to C, C++, COBOL and Enterprise PL/I for z/OS.

The Language Environment prelinker performs mapping of names, manages writable static areas, collects initialization information, and combines the object modules that form an application into a single object module that can be link-edited or loaded for execution.

Note:
The prelink step in creating an executable program can be eliminated. The binder is able to directly receive the output of the C, C++, COBOL, and Enterprise PL/I for z/OS compilers, thus eliminating the requirement for the prelink step. The advantage of using the binder is that the resulting executable program is fully rebindable.

IBM intends to stabilize the prelinker. The prelinker was designed to process long names and support constructed reentrancy in earlier versions of the C, C++, COBOL, and PL/I compilers, and the Language Environment-conforming assembler, on the MVS and OS/390 operating systems. The prelinker provides output that is compatible with the linkage editor, shipped with the program management binder.

The program management binder is designed to include the function of the prelinker, the linkage editor, the loader, and a number of APIs to manipulate the program object. Its functionality delivers a high level of compatibility with the prelinker and linkage editor, but provides additional functionality in some areas.

Further enhancements will not be made to the prelinker utility. Enhancements will be made only to the program management binder, to position the program management binder as the strategic tool for program object manipulation.

For information on how to use the binder, see z/OS MVS Program Management: User's Guide and Reference and z/OS MVS Program Management: Advanced Facilities.

For information on how to build and use DLLs, see Chapter 4, “Building and using dynamic link libraries (DLLs),” on page 37.

Which programs need to be prelinked

The prelink step is required when an executable program is to reside in a PDS, or if it utilizes the system programming facilities of C. When the executable is to reside in a PDSE or HFS, the prelink step may be eliminated since the binder can handle the output of the C, C++, COBOL, and Enterprise PL/I for z/OS compilers. If the link-edit process is performed by the linkage editor then the prelink step is required.

You should not use the pre-linker with XPLINK programs because XPLINK programs require the GOFF binder format and GOFF is not supported by the pre-linker. Also, the z/OS XL C/C++ compiler creates GOFF object code when the XPLINK compiler option is specified. When bound, the objects must reside in PDSEs or the HFS.

The following list identifies programs which may need to be prelinked before the link-edit step of creating an executable program.

- Modules which must be processed with the linkage editor rather than the binder
- Modules which must be stored in a PDS rather than in a PDSE
- Programs which utilize the system programming facilities of C.
- Non-XPLINK C programs compiled with any of the following compiler options:
  - RENT
  - LONGNAME
  - DLL
• Non-XPLINK C++ programs
• COBOL programs compiled with any of the following compiler options:
  – DLL
  – PGMNAME(LONGMIXED)
  – PGMNAME(LONGUPPER)
• COBOL programs that use object-oriented extensions
• COBOL programs containing class definitions or the INVOKE statement
• Enterprise PL/I for z/OS programs
• Programs compiled to run under z/OS UNIX

Only C object modules that do not refer to writable static, do not contain the LONGNAME option, and do not contain DLL code can be processed by the linkage editor. You do not need to prelink naturally reentrant programs. For more information, see “Making your C/C++ program reentrant” on page 121.

If you need to link-edit together object modules and load modules, prelink the object modules through the prelinker in a single step, and then link-edit with the load modules in a separate link-edit step. This is because the prelinking process can only process object modules.

**What the prelinker does**

The prelinker performs the following functions:

• Collects information for runtime initialization, including data initialization for C/C++, constructor/destructor calls for static objects in C++, and DLL initialization information.
• For C object modules compiled with RENT, C++ programs, Enterprise PL/I for z/OS programs, or COBOL programs with OO extensions, the prelinker:
  – Combines writable static initialization information
  – Assigns relative offsets to objects in writable static storage
  – Removes writable static name and relocation information
• For programs containing longnames, such as C programs compiled with LONGNAME, C++ programs, Enterprise PL/I for z/OS programs, and COBOL programs compiled with PGMNAME(LONGMIXED) or PGMNAME(LONGUPPER), the prelinker maps LONGNAME option to SHORTNAME option on output.
• For programs that use DLLs, the prelinker:
  – Generates a function descriptor in writable static for each DLL referenced function
  – Generates a variable descriptor for each DLL referenced variable
  – Generates an IMPORT control statement for each exported function and variable
  – Generates internal information for the load module that describes symbols that are exported to and imported from other load modules
  – Combines static DLL initialization information
  – Uses longnames to resolve exported and imported symbols

**Prelinking process**

Input to the prelinker includes the following:

• Primary input: those data sets and DLL definition sidedecks that are allocated to SYSIN. If you are creating an application that imports symbols from DLLs, you must provide the definition sidedeck for each DLL in SYSIN.
• Secondary input: input that is processed from SYSLIB, which contains object module libraries that are used for automatic library calls.
Input that is specified in one or more INCLUDE control statements that are processed as primary and secondary input.

An attempt is made to read the DD or member of the DD (whichever is specified). This request is resolved if the read is successful.

If you are exporting symbols, the prelinker creates a definition sidedeck. After the prelinker processes all its input, it puts the prelinked output object module into SYSMOD. If a definition sidedeck was generated, it is put into SYSDEFSD and is a sequential data set or a PDS member. The linking process then begins when the linkage editor takes its primary input from SYSLIN, which refers to the prelinked object module data set.

The IBM-supplied cataloged procedures and REXX EXECs for C/C++ use the DLL versions of the IBM-supplied class libraries by default; the IBM-supplied class libraries definition sidedeck data set, SCLBSID, is included in the SYSIN concatenation.

If you are statically linking the relevant C/C++ class library object code, you must:

- Override the PLKED.SYSLIB concatenation to include the SCLBCPP data set, and
- Override the PLKED.SYSIN concatenation to exclude the SCLBSID data set.

Figure 109 on page 484 shows an overview of the basic prelinking process.
References to currently unresolved symbols (unresolved external references)

If, during the automatic library call, a symbol is not the name of an existing member of an object data set, the symbol can subsequently be defined if a function or variable with the same name is encountered. Unresolved requests generate error or warning messages to the prelinker map.
Writable static references that are not resolved by the prelinker cannot be resolved later. Only the prelinker can be used to resolve writable static. The output object module of the prelinker should not be used as input to another prelink.

If you are building an application that imports symbols from a DLL, you must include the definition sidedeck produced by the prelinker when the DLL was built as input to the prelink step of your application.

If the symbol is an L-name that was not resolved by automatic library call and for which a RENAME statement with the SEARCH option exists, the symbol is resolved under the S-name on the RENAME statement by automatic library call. See “RENAME control statement” on page 489 for a complete description of the RENAME control statement.

Unresolved references or undefined writable static objects often result if the prelinker is given input object modules produced with a mixture of RENT/NORENT or LONGNAME/NOLONGNAME or DLL options. For more information on avoiding unresolved references in a DLL or in an application that imports symbols from a DLL, see z/OS XL C/C++ User’s Guide.

Processing the prelinker automatic library call

The following hierarchy is used to resolve a referenced and currently undefined symbol. In all cases, the symbol is only defined if it is contained in the input from this process or in other future input:

- The undefined name is an S-name, for example SNAME.
  - If the NONCAL command option is in effect, the partitioned data sets concatenated to SYSLIB are searched in order as follows:
    - If the data set contains a C370LIB-directory created using the Object Library Utility, and the C370LIB-directory shows that a defined symbol by that name exists, the member of the PDS containing that symbol is read.
    - If the data set does not contain a C370LIB-directory created using the Object Library Utility and the reference is not to static external data, the member or alias, with the same name as SNAME, is read.
- The undefined name is an L-name.
  - If the NONCAL command option is in effect, the partitioned data sets concatenated to SYSLIB are searched. If the data set contains a C370LIB-directory created using the Object Library Utility, and the C370LIB-directory shows that a defined symbol by that name exists, the member of the PDS indicated as containing that symbol is read.

For more information about the Object Library Utility, see z/OS XL C/C++ User’s Guide or Appendix E, “Object library utility,” on page 513.

Language Environment prelinker map

The Language Environment prelinker produces a listing file called the prelinker map when you use the MAP prelinker option (which is the default). As the following example shows, the prelinker map contains several individual sections that are only generated if they are applicable.

========================================================================
|                            Prelinker Map                               |
| CPLINK:5647A01 V2 R9 M00 IBM Language Environment 2000/01/20 13:45:16 |
| Command Options. . . . . : NONCAL   NOMEMORY ER       DUP      MAP      |
| : NOOMVS   NOUPCASE DYNAM                                           |
|========================================================================
|                      Object Resolution Warnings                        |
| WARNING EDC4015: Unresolved references are detected:           |
| CEESTART @TRGLOR CEESG003                                        |
Prelinking applications

The numbers in the following text correspond to the numbers shown in the map.
Heading
The heading is always generated and contains the product number, the library release number, the library version number, the date and the time the prelink step began, followed by a list of the prelinker options in effect for the step.

Object Resolution Warnings
This section is generated if objects remained undefined at the end of the prelink step or if duplicate objects were detected during the step. The names of the applicable objects are listed.

File Map
This section lists the object modules that were included in input. An object module consisting only of RENAME control statements, for example, is not shown. Also provided in this section are source origin (*ORIGIN), name (FILE_NAME), and identifier (FILE_ID) information. *ORIGIN indicates that the object module came from primary input because of:

- An INCLUDE control statement in primary or secondary input.
- A RENAME control statement.
- The resolution of L-name library references.
- The object module was internal and self-generated by the prelink step.

The FILE_ID can be found in other sections and is used as a cross-reference to the object module.

The FILE_NAME can be either the data set name and, if applicable, the member name, or the ddname and, if applicable, the member name.

If you are prelinking an application that imports variables or functions from a DLL, the variable descriptors and function descriptors are defined in a file called *** DESCRIPTORS ***. This file has an origin of internal.

Writable Static Map
This section is generated if an object module was encountered that contains defined static external data. This area also contains variable descriptors for any imported variables and, if required, function descriptors. This section lists the names of such objects, their lengths, their relative offset within the writable static area, and a FILE_ID for the file containing the object's definition.

Imported variables and DLL-referenced functions have angular brackets (<>) around their names in this section.

Load Module Map
This section is generated if the application imports symbols from other load modules. This section lists the names of the load modules.

Import Symbol Map
This section lists the symbols that are imported from other load modules. These otherwise unresolved DLL references are resolved through IMPORT control statements. It describes the type of symbol, that is, D (variable) or C (function). It also lists the file ID of the object module containing the corresponding IMPORT control statements, the module ID of the load module on that control statement, and the symbol name.

A DLL application would generate this section.

Export Symbol Map
This section lists the symbols generated by an object module that exports symbols. It describes the type of symbol, that is, D (variable) or C (function). It also lists the file ID of the object where the symbol is defined and the symbol name. Only externally defined data objects in writable static or externally defined functions can be exported.

Code that is compiled with the C, C++, or COBOL EXPORTALL compiler option or C/C++ code containing the #pragma export directive generates an object module that exports symbols.

Note: The export symbol map will NOT be produced if the NODYNAM option is in effect.

ESD Map of Defined and Longnames
This section lists the names of external symbols that are not in writable static. It also shows a mapping of input L-names to output S-names.
If the object is defined, the FILE ID indicates the file that contains the definition. Otherwise, this field is left blank. For any name, the input name and output S-name are listed. If the input name is an L-name, the rule used to map the L-name to the S-name is applied. If the name is not an L-name, this field is left blank.

Control statement processing

The only control statements processed by the prelinker are IMPORT, INCLUDE, LIBRARY, and RENAME. The remaining control statements are left unchanged until the link-edit step.

The control statements can be placed in the input stream or stored in a permanent data set.

Note: If you cannot fit all of the information on one control statement, you can use one or more continuations. The L-name, for example, can be split across more than one statement. Continuations are enabled by placing a nonblank character in column 72 of the statement that is to be continued. They must begin in column 16 of the next statement.

IMPORT control statement

The prelinker processes IMPORT statements, but does not pass them on to the link step. The IMPORT control statement has the following syntax:

```
IMPORT
  CODE dll-name function
  DATA dll-name variable
```

**dll-name**
The name or alias of the load module for the DLL. The maximum length of an alias is 8 characters. The *dll-name* can also be a z/OS UNIX name; it must be enclosed in apostrophes if special characters, such as apostrophes or blanks, appear in the *dll-name*.

**variable**
An exported variable name; it is a mixed-case longname. Use a nonblank character in column 72 of the card to indicate a continuation and begin the next line in column 16.

**function**
An exported function name; it is a mixed-case longname. Use a nonblank character in column 72 of the card to indicate a continuation and begin the next line in column 16.

INCLUDE control statement

The INCLUDE control statement has the following syntax:

```
INCLUDE
  filename
  ddname (member)
```

**filename**
The name of the file to be included.

**ddname**
A ddname associated with a file to be included.

**member**
The member of the DD to be included.

The prelinker processes INCLUDE statements like the DFSMS linkage editor does with the following exceptions:
• INCLUDEs of identical member names are not allowed.
• INCLUDEs of both a ddname and a member from the same ddname are not allowed. The prelinker ignores the second INCLUDE.

**LIBRARY control statement**

The LIBRARY control statement has the following syntax:

```
LIBRARY
  name
  (member)
  (external)
```

`name`

The ddname defining a library. The ddname can point to an archive file in the z/OS UNIX file system if the OE option is specified, or a PDS object library. The PDS object library can be a concatenation of one or more libraries created with or without the Object Library Utility.

`member`

The name or alias of a member of the specified library. Because both S-names and L-names can be specified, case distinction is significant.

Automatic library calls search the library and each subsequent library in the concatenation, if necessary, for the name instead of searching the primary input.

If you specify the OMVS or OE prelinker option, the only form of the LIBRARY card accepted by the prelinker is `LIBRARY ddname`, which specifies a library to search immediately for autocall.

`external`

An external reference that could be unresolved after primary input processing. This external reference will not be resolved by an automatic library call. Because both S-names and L-names can be specified, case distinction is significant.

The LIBRARY control statement is removed and not placed in the prelinker output object module; the system linkage editor does not see the LIBRARY control statement.

**RENAME control statement**

The RENAME control statement has the following syntax:

```
RENAME
  L-name
  S-name
  SEARCH
```

`L-name`

The name of the input L-name to be renamed on output. All occurrences of this L-name are renamed.

`S-name`

The name of the output S-name to which the L-name will be changed. This name can be at most 8 characters and case is respected.

`SEARCH`

An optional parameter specifying that if the S-name is undefined, the prelinker searches by an automatic library call for the definition of the S-name. SEARCH is not supported under z/OS UNIX.

The RENAME control statement is processed by the prelinker and can be used for several purposes:

• To explicitly override the default name given to an L-name when an L-name is mapped to an S-name.

You can explicitly control the names presented to the system linkage editor so that external variable and function names are consistent from one linkage editor run to the next. This consistency makes it easier to recognize control section and label names that appear in system dumps and linkage editor listings.
Another mapping rule (described in “Mapping L-Names to S-Names” on page 490) can provide the suitable name, but if you need to replace the linkage editor control section, you need to maintain consistent names.

- To explicitly bind an L-name to an S-name. This binding might be necessary when communicating with objects from other language and assembler processors, because these processors generate only S-names.
- A RENAME control statement cannot be used to rename a writable static object because its name is not contained in the output from the prelinker.

RENAME control statements can be placed before, between, or after other control statements or object modules. An object module can contain only RENAME statements. Also, RENAME statements can be placed in input that is included because of other RENAME statements.

Usage notes

- A RENAME statement is ignored if the L-name is not encountered in the input.
- A RENAME statement for an L-name is valid provided all of the following are true:
  - The L-name was not already mapped because of a rule that preceded the RENAME statement rule in the hierarchy described in “Mapping L-Names to S-Names” on page 490.
  - The L-name was not already mapped because of a previous valid RENAME statement for the L-name.
  - The S-name is not itself an L-name. This rule holds true even if the S-name has its own RENAME statement.
  - A previous valid RENAME statement did not rename another L-name to the same S-name.
  - Either the L-name or the S-name is not defined. Either the L-name or the S-name can be defined, but not both. This rule holds true even if the S-name has its own RENAME statement.

Mapping L-Names to S-Names

The output object module of the prelinker can be used as input to a system linkage editor.

Because system linkage editors accept only S-names, the Language Environment prelinker maps L-names to S-names on output. S-names are not changed. L-names can be up to 160 (COBOL for OS/390 & VM and COBOL for MVS & VM), 255 (z/OS XL C/C++), or 1024 (z/OS XL C++) characters in length; truncation of the L-names to the 8-character S-name limit is therefore not sufficient because collisions can occur.

The Language Environment prelinker maps a given L-name to a S-name according to the following hierarchy:

1. **C/C++ only:** If any occurrence of the L-name is a reserved runtime name, or was caused by a `#pragma map` or `#pragma CSECT` directive, then that same name is chosen for all occurrences of the name. This name must not be changed, even if a RENAME control statement for the name exists. For information on the RENAME control statement, see “RENAME control statement” on page 489.

2. If the L-name was found to have a corresponding S-name, the same name is chosen. For example, `D0TOTALS` is coded in both a C and assembler program. This name must not be changed, even if a RENAME statement for the name exists. This rule binds the L-name to its S-name.

3. If a valid RENAME statement for the L-name is present, the S-name specified on the RENAME statement is chosen.

4. If the name corresponds to a Language Environment function or library object for which you did not supply a replacement, the name chosen is the truncated, uppercased version of the L-name library name (with _ mapped to @).

   The S-name is not chosen, if either:
   - A valid RENAME statement renames another L-name to this S-name. For example, the RENAME statement `RENAME mybigname PRINTF` would make the library `printf()` function unavailable if mybigname is found in input.
Another L-name is found to have the same name as the S-name. For example, explicitly coding and referencing `PRINTF` in the C source program would make the library `sprintf()` function unavailable.

Avoid such practices to ensure that the appropriate Language Environment function is chosen.

5. If the UPCODE option is specified, names that are 8 characters or fewer are changed to uppercase (with _ mapped to @). Names that begin with IBM or CEE will be changed to IB$, and CE$, respectively. Because of this rule, two different names can map to the same name. You should therefore use the UPCODE option carefully. A warning message is issued if a collision is found, but the names are still mapped.

6. If none of the above rules apply, a default mapping is performed. This mapping is the same as the one the compiler option NOLONGNAME uses for external names, taking collisions into account. That is, the name is truncated to 8 characters and changed to uppercase (with _ mapped to @). Names that begin with IBM or CEE will be changed to IB$ and CE$, respectively. If this name is the same as the original name, it is always chosen. This name is also chosen if a name collision does not occur. A name collision occurs if either

- The S-name has already been seen in any input, that is, the name is not new.
- After applying this default mapping, the same name is generated for at least two, previously unmapped, names.

If a collision occurs, a unique name is generated for the output name. For example, the name @ST00033 is manufactured.

z/OS XL C/C++: A program that is compiled with the NOLONGNAME compiler option and link-edited, except for collisions, library renames, and user renames, presents the linkage editor with the same names as when the program is compiled with the LONGNAME option and processed by the prelinker.

### Starting the prelinker under batch and TSO/E

The following topics describe how to start the prelinker under batch and TSO/E. For more information about prelinking z/OS XL C++ applications under batch, see z/OS XL C/C++ User’s Guide.

#### Under batch

The prelinker is invoked by the following cataloged procedures:

- **CBCCL**
  - C++ compile, prelink, and link
- **CBCCLG**
  - C++ compile, prelink, link, and run
- **CBCL**
  - C++ prelink and link
- **CBCLG**
  - C++ prelink, link, and run
- **EDCPL**
  - C prelink and link
- **EDCCPLG**
  - C compile, prelink, link, and run.
- **IGYWCP**
  - COBOL compile, prelink, link, and run
- **IGYWPL**
  - COBOL prelink and link
- **IGYWCPG**
  - COBOL compile, prelink, load, and run
IBMZCPG
Enterprise PL/I for z/OS compile, prelink, and load/run using the loader

IBMZCPL
Enterprise PL/I for z/OS compile, prelink, and link

IBMZCPLG
Enterprise PL/I for z/OS compile, prelink, link, and run

For more information about using these procedures, see z/OS XL C/C++ User’s Guide, the appropriate version of the COBOL programming guide in the COBOL library at Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733), or Enterprise PL/I for z/OS Programming Guide.

Under TSO/E

The Language Environment prelinker is started under TSO/E through an IBM-supplied CLIST called CPLINK, which invokes the prelinker and creates an executable module. If you want to create a reentrant C/C++ load module, link-edit C/C++ or COBOL object modules using long names, or create a DLL application, you must use CPLINK instead of the TSO/E LINK command.

The CPLINK command has the following syntax:
OBJ
Specifies an input data set name. This is a required parameter. Each input data set must be one of the following:
- A C object module compiled with the RENT or LONGNAME compiler options
- A C object module that has no static external data
- A COBOL object module

POPT
Specifies a string of prelink options. The prelinker options available for CPLINK are the same as for batch. For example, if you want the MAP option to be used by the prelinker, specify the following:

```cplink obj('dsname') popt('map')...```

When the prelink MAP option is specified (as opposed to the link option MAP), the prelinker produces a file showing the mapping of static external data. This map shows name, length, and address information. Any unresolved references or duplicate symbols during the prelink step are displayed in the map.

PLIB
Specifies the library names used by the prelinker for the automatic library call facility.

LOPT
Specifies a string of linkage editor options. For example, if you want the prelink utility to use the MAP option and the linkage editor to use the NOMAP option, use the following CLIST command:

```cplink obj('dsname') popt('map') lopt('nomap...')```

LIB
Specifies any additional library or libraries used by the TSO/E LINK command to resolve external references. These libraries are appended to the default language library functions.

LOAD
Specifies an output data set name. If you do not specify an output data set name, a name is generated for you. The name generated by the CLIST consists of your user prefix followed by CPOBJ.LOAD(TEMPNAME).

Examples
In the following example, your user prefix is RYAN, and the data set containing the input object module is a partitioned data set called RYAN.C.OBJ(INCCOMM). This example will generate a prelink listing without using the automatic call library. After the call, the load module is placed in a partitioned data set called RYAN.CPOBJ.LOAD(TEMPNAME) and the prelink listing is placed in a sequential data set called RYAN.CPOBJ.RMAP.

```cplink obj('C.OBJ(INCCOMM)')```

In the following examples, assume that your user prefix is DAVE, and the data set containing the input object module is a partitioned data set called DAVE.C.OBJ(INCPYRL). This example will not generate a
prelink listing, and the automatic call facility will use the library HOOVER.LIB.SUB. The load module is
placed in the partitioned data set DAVE.TBD.LOAD(MOD).

```
//**-----------------------------------------------
//  Prelink and link 'DAVE.C.OBJ(INCPYRL)'  
//**-----------------------------------------------
//P0014001  EXEC EDCPL,
//    INFILE='DAVE.C.OBJ(INCPYRL)',
//    OUTFILE='DAVE.TBD.LOAD(MOD),DISP=SHR',
//    PPARM='NOMAP,NONCAL',
//    PLIB='HOOVER.LIB.SUB',
//    LPARM='AMODE(31),RMODE(ANY)'   
//=-------------------------------------------------------
```

Figure 110: Example of prelinking under batch

```
CPLINK OBJ('''DAVE.C.OBJ(INCPYRL)''')
    POPT('NOMAP,NONCAL')
    PLIB('''HOOVER.LIB.SUB''')
    LOAD('TBD.LOAD(MOD)')
```

Figure 111: Example of prelinking under TSO/E

**Using the CXXBIND EXEC under TSO/E**

For a description of using the CXXBIND EXEC to build a C++ executable program without using the prelink
step, see *z/OS XL C/C++ User's Guide*.

**Using the CXXMOD EXEC under TSO/E**

This topic describes how to prelink and link your C++ or COBOL program by invoking the CXXMOD EXEC.
This exec creates an executable module. The syntax for the CXXMOD EXEC is:
OBJ
You must always specify the input data set names on the OBJ keyword parameter. Each input data set must be a C/C++, COBOL, or assembler object module.

If the high-level qualifier of an object data set is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

POPT
Prelinker options can be specified using the POPT keyword parameter. If the MAP prelink option is specified, a prelink map will be written to the data set specified under the PMAP keyword parameter. For more details on generating a prelink map, see the description of the PMAP option.
LOPT
Linkage editor options can be specified using the LOPT keyword parameter. For details on how to generate a linkage editor listing, see the option LIST.

PLIB
The library names that are to be used by the automatic call library facility of the prelinker must be specified on the PLIB keyword parameter. The default library used is the C++ base library, CEE.SCEECPP.

The default library names are not added if library names are specified with the PLIB keyword parameter.

If the high-level qualifier of a library data set is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

LIB
If you want to specify libraries for the link-edit step to resolve external references, use the LIB keyword parameter. The default library used is the Language Environment library, CEE.SCEELKED.

The default library names are not added if library names are specified with the LIB keyword parameter.

If the high-level qualifier of a library data set is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

PMOD
If you want to keep the output prelinked object module, specify the data set it should be placed in using the PMOD keyword parameter. The default action is to create a temporary data set and erase it after the link-edit is complete.

If the high-level qualifier of the output prelinked object module is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

LOAD
To specify where the resultant load module should be placed, use the LOAD keyword parameter.

If the high-level qualifier of the load module is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

LIST
To specify where the linkage editor listing should be placed, use the LIST keyword parameter.

If the high-level qualifier of the linkage editor listing is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

PMAP
To specify where the prelinker map should be placed, use the PMAP keyword parameter.

If the high-level qualifier of the prelinker map is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

PDEF
To specify where the generated IMPORT control statements should be placed by the prelinker.

If the high-level qualifier of the prelinker map is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

### Prelinker options

Table 94 on page 497 describes the prelinker options.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DLLNAME</strong> (<em>dll-name</em>)</td>
<td>DLLNAME specifies the DLL name that appears on generated IMPORT control statements. If you specify the DLLNAME option, the DLL name is set to the value listed on the option. If you do not specify DLLNAME, the DLL name is set to the name that appeared on the last NAME control statement that was processed. If there are no NAME control statements, and the output object module of the prelinker is a PDS member, the DLL name is set to the name of that member. Otherwise, the DLL name is set to the value TEMPNAME, and the prelinker issues a warning. The default is DLLNAME.</td>
</tr>
<tr>
<td>**DUP</td>
<td>NODUP**</td>
</tr>
<tr>
<td>**DYNAM</td>
<td>NODYNAM**</td>
</tr>
<tr>
<td>**ER</td>
<td>NOER**</td>
</tr>
<tr>
<td>**MAP</td>
<td>NOMAP**</td>
</tr>
<tr>
<td>**MEMORY</td>
<td>NOMEMORY**</td>
</tr>
<tr>
<td>**NCAL</td>
<td>NONCAL**</td>
</tr>
</tbody>
</table>
### Table 94: Prelinker options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>**OE</td>
<td>NOOE**</td>
</tr>
<tr>
<td>**OMVS</td>
<td>NOOMVS**</td>
</tr>
<tr>
<td>**UPCASE</td>
<td>NOUPCASE**</td>
</tr>
</tbody>
</table>
Appendix B. EXEC DLI and CALL IMS Interfaces

There are two major approaches to accessing DL/I databases on IMS. This topic describes the interfaces that are supported in the various environments, specifically IMS, and CICS.

- The EXEC DLI approach.
- The CALL IMS approach, which includes interfaces, such as: PLITDLI, CBLTDLI, and CTDLI.

Either an IMS library or a CICS library should be present when linking an application. If both libraries are available, link-edit errors might occur.

If you are using ILC in CICS DL/I applications, EXEC CICS DLI and CALL xxxTDLI can only be used in programs with the same language as the main program. For details on using ILC under CICS, see CICS Transaction Server for z/OS (www.ibm.com/support/knowledgecenter/SSGMGV/welcome.html).

The following table lists the IMS and CICS support for various user interfaces to DL/I databases.

<table>
<thead>
<tr>
<th>User interface</th>
<th>IMS supported</th>
<th>CICS supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL CEETDLI</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CALL AIBTDLI</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CALL CBLTDLI (COBOL)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CALL PLITDLI (PL/I)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ctdli() (C)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CALL ASMTDLI (non-PL/I)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CALL ASMTDLI (PL/I)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EXEC DLI (non-C)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EXEC DLI (C)</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
EXEC DLI and CALL IMS interfaces
Appendix C. Guidelines for writing callable services

If you want to write services similar in form and description to Language Environment callable services, follow the guidelines listed below.

- Callable service parameters must follow the data type descriptions outlined in z/OS Language Environment Programming Reference.
- Argument passing is by one level of indirection, either “by reference” or “by value”. See “Passing arguments between routines” on page 114 for these argument passing styles.
- Avoid the use of operating system services and macros. Use Language Environment services whenever possible.
- Always use the prototype definition or the entry declaration whenever possible.
- Avoid using the CEE3SPM callable service (see z/OS Language Environment Programming Reference). CEE3SPM can change the condition handling semantics of the HLLs supported by Language Environment.
- Language Environment assumes the following defaults for character strings:
  - For input arguments, a length-prefixed string (with the length of the 2-byte prefix not included in the length value)
  - For output arguments, a fixed-length string of 80 bytes, padded on the right with blanks as necessary.
- Allow a feedback code area to be optionally passed as the last parameter to the callable service. The feedback code must be a FEED_BACK data type and conform to the layout described in Chapter 18, “Using condition tokens,” on page 233.
- If omitted arguments are permitted by the HLL, a zero or NULL pointer must be used to indicate the omitted parameter in the parameter list that is passed to the callable service. For example:

```
address of parm1
address of parm2
0
```

The last parameter passed in the list must have the high-order bit on to indicate that it is last. If the last parameter is omitted, the zero value that the user passes in the parameter list must have the high-order bit on, for example, X’80000000’. Therefore, you must allow the user of the callable service to check for this bit when the last parameter passed to the service is omitted.

- When documenting callable services, follow the same general format used to document each of the callable services in this information. Each callable service description should contain (in this order):
  - A general description of what the service does.
  - A diagram indicating the syntax of the call to the service.
  - A complete description of each callable service parameter and an identification of the required data type.
  - A list of possible feedback codes that can be returned by the service to its caller.
  - Usage notes that provide additional needed information to the user, such as a list of related callable services.
  - An example or examples of usage.
Appendix D. Operating system and subsystem parameter list formats

This topic describes the various formats of parameters passed to and from operating systems and subsystems. In most cases, you do not need to know these formats in order to pass or receive parameters in your application. For cases in which you want to directly access the parameter list that is passed, the format and contents of the parameter list are shown later in this section.

There are additional considerations depending on whether the main routine is in the C, C++, COBOL, or PL/I language.

C and C++ parameter passing considerations

C and C++ generally support a single character string as a parameter to a main routine. They parse the string into tokens that are accessed by the argc and argv parameters of the main function.

In addition, there are alternate styles of passing a set of parameters to the main routine, for example: as a single value, a pointer to a value, or a pointer to a list of values. In these cases, the set of parameters is not parsed. It is assumed that the invoker of the application (for example, the operating system) has stored the address of the set of parameters in register 1 before entry into the main routine. Depending on how the parameters are passed, register 1 points on entry to the entities illustrated in Figure 112 on page 503:

Style 1: Register 1 contains parameter value

Register 1 = parameter value

Style 2: Register 1 contains pointer to parameter value

Register 1 = pointer → parameter value

Style 3: Register 1 contains pointer to array of pointers to parameter values

Register 1 = pointer → (pointer0 → value0)
(pointer1 → value1)
(pointer2 → value2)
...

(pointern → valuen)

Figure 112: Alternate C/C++ parameter passing styles

The first arrangement in Figure 112 on page 503 can be used only for parameters that are integers.

A C main routine elects to use one of the styles shown in Figure 112 on page 503 by specifying the PLIST(OS) runtime option in #pragma runopts (see “C PLIST and EXECOPS interactions” on page 505); a C++ routine elects to use one of the styles with the PLIST(OS) compiler option (see “C++ PLIST and EXECOPS interactions” on page 507). The main routine must know which parameter style to expect. When PLIST(OS) is specified, C or C++ makes the parameter list available through a pair of macros; code them in your main routine to determine which parameter list style your routine receives:
__R1 of type void *
__R1 contains the value that is in register 1 on entry into the main routine. It provides access to the
parameters when they are passed according to the first two styles shown in Figure 112 on page 503.

__osplist of type void **
__osplist acts as an array of pointers to parameters. It is derived from __R1 and provides access to
the parameters when they are passed according to the third style shown in Figure 112 on page 503.
You must include the header file stdlib.h when using __osplist.

The third style is also supported for certain macros and functions (for example, __pcblist and
__csplist for invokers IMS and Cross System Product). __osplist is a generalization of the more
specialized __pcblist and __csplist macros; it can be used in their place or in cases where they do
not apply.

Figure 113 on page 504 illustrates how these macros can be used to access items in the three alternate
parameter arrangements.

Style 1:
Register 1 = __R1

Style 2:
Register 1 = __R1  →  *__R1

Style 3:
Register 1 = __R1  →  (__osplist[0]  →  *__osplist[0])
               ( __osplist[1]  →  *__osplist[1])
               ( __osplist[2]  →  *__osplist[2])

               ...

               ( __osplist[n]  →  *__osplist[n])

Figure 113: Accessing parameters using macros __R1 and __osplist

Suitable casting and dereferencing are required when using these macros, as shown in Figure 114 on
page 504, according to the parameter passing style in use.

Style 1:
parm    = (int) __R1;  (restricted to integer types)

Style 2:
parm_ptr = (float *) __R1
parm    = * (float *) __R1;

Style 3:
parm0_ptr = (float *) __osplist[0];
parm0    = * (float *) __osplist[0];

Figure 114: Examples of casting and dereferencing
C PLIST and EXECOPS interactions

You can use C `#pragma runopts` to specify to the C compiler a list of options to be used at run time. Two of the options of `#pragma runopts` affect the format of the argument list passed to the application on initialization: EXECOPS and PLIST.

EXECOPS allows you to specify runtime options on the command line or in JCL at application invocation. NOEXECOPS indicates that runtime options cannot be so specified. When the EXECOPS runtime option is specified under MVS, Language Environment alters the MVS parameter list format: Language Environment removes any runtime options that are present.

PLIST indicates in what form the invoked routine should expect the argument list. You can specify PLIST with the following values under Language Environment:

**HOST**

The argument list is assumed to be a character string. The string is located differently under various systems as follows:

- Under TSO, if a CPPL is detected, Language Environment gets the string from the command buffer.
- Under TSO, if a CPPL is not detected, Language Environment assumes a halfword-prefixed string in the MVS format.
- Under MVS, Language Environment uses the halfword-prefixed string.

**OS**

The inbound parameter list is assumed to be in an MVS linkage format in which register 1 points to a parameter address list. No runtime options are available. Register 1 is not interrogated by Language Environment.

The PLIST(HOST) setting allows the object to execute under MVS (assuming a halfword-prefixed string), or under TSO (using the CPPL or the MVS-format parameter list). Specify PLIST(HOST) to default to the argument list format for the operating system under which your application is running.

Although Language Environment supports the MVS, IMS, and TSO suboptions of PLIST for compatibility, use of PLIST(HOST) is recommended. There are some exceptions to this guideline:

**Preinitialization**

- PLIST(MVS) is supported for compatibility with pre-Language Environment C preinitialization programs.

**CICS**

- If you are running a CICS application compiled under the pre-Language Environment-conforming version of C, PLIST(HOST), the default, is assumed regardless of the actual PLIST setting. If you are running a CICS application compiled with a Language Environment-conforming C compiler, specify PLIST(OS).

**TSO**

- TSO command processors that require access to the full CPPL must specify PLIST(OS).

The EXECOPS, NOEXECOPS, and PLIST options can alter the format of the argument list passed to your application, depending on the combination of options specified. The setting of EXECOPS determines whether Language Environment looks for runtime parameters in the inbound parameter list. The effects of the interactions of these options under the various operating systems and subsystems are summarized in Table 96 on page 506:
### Table 96: Interactions of C PLIST and EXECOPS

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Method of invocation</th>
<th>PLIST suboption</th>
<th>EXECOPS suboption (default)</th>
<th>arg/argv</th>
<th>__R1/__osplist and PCBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVS</td>
<td>EXEC PGM=, PARM= &lt;runtime options&gt; / &lt;user args&gt;</td>
<td>HOST</td>
<td>Yes. &lt;runtime options&gt; honored</td>
<td>argc = number of tokenized args in &lt;user args&gt; argv[0...argc-1] = tokenized args in &lt;user args&gt;</td>
<td></td>
</tr>
<tr>
<td>MVS</td>
<td>EXEC PGM=, PARM= &lt;runtime options&gt; / &lt;user args&gt;</td>
<td>HOST</td>
<td>No. &lt;runtime options&gt; ignored</td>
<td>argc = number of tokenized args in the entire PARM string, that is, &lt;runtime options&gt; / &lt;user args&gt; argv[0...argc-1] = tokenized args in the entire PARM string</td>
<td></td>
</tr>
<tr>
<td>MVS</td>
<td>Assembler calls C module with pre-Language Environment preinitialization PLIST with runtime options specified in the PLIST</td>
<td>MVS</td>
<td>Yes. &lt;runtime options&gt; honored</td>
<td>argc/argv = &lt;argc,argv&gt; structure specified in the preinitialization PLIST</td>
<td></td>
</tr>
<tr>
<td>MVS</td>
<td>Assembler calls C module with pre-Language Environment preinitialization PLIST with runtime options specified in the PLIST</td>
<td>MVS</td>
<td>No. &lt;runtime options&gt; ignored</td>
<td>argc/argv = &lt;argc,argv&gt; structure specified in the preinitialization PLIST</td>
<td></td>
</tr>
<tr>
<td>MVS</td>
<td>Driver link to C main passing noncharacter parameter list</td>
<td>OS</td>
<td>n/a</td>
<td>argc=1 argv[0] = name of C main program module</td>
<td>Access register 1 through __osplist macro as defined in stdlib.h</td>
</tr>
<tr>
<td>TSO</td>
<td>CALL, LOADGO, execute module on TSO command line passing &lt;runtime options&gt; / &lt;user args&gt;</td>
<td>HOST</td>
<td>Yes. &lt;runtime options&gt; honored</td>
<td>argc = number of tokenized args in &lt;user args&gt; argv[0...argc-1] = tokenized args in &lt;user args&gt;</td>
<td></td>
</tr>
<tr>
<td>Operating system</td>
<td>Method of invocation</td>
<td>PLIST suboption</td>
<td>EXCECOPS (default)</td>
<td>arg/argv</td>
<td>__R1/__osplist and PCBs</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------</td>
<td>-----------------</td>
<td>-------------------</td>
<td>----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>TSO</td>
<td>CALL, LOADGO, execute module on TSO command line passing &lt;runtime options&gt; / &lt;user args&gt;</td>
<td>HOST</td>
<td>No. &lt;runtime options&gt; ignored</td>
<td>argc = number of tokenized args in &lt;runtime options&gt; / &lt;user args&gt; argv[0...argc-1] = tokenized args in &lt;user args&gt;</td>
<td></td>
</tr>
<tr>
<td>TSO</td>
<td>CALL</td>
<td>OS</td>
<td>n/a</td>
<td>argc=1</td>
<td>argv[0] = name of module</td>
</tr>
<tr>
<td>IMS</td>
<td>Invoke C main module</td>
<td>OS or IMS Specifying ENV(IMS) also.</td>
<td>n/a</td>
<td>argc=1</td>
<td>argv[0] = name of C main module or null if the #pragma runopts(PLIST(IMS)) is present in the source</td>
</tr>
<tr>
<td>CICS</td>
<td>Invoke C main module</td>
<td>n/a</td>
<td>n/a</td>
<td>argc=1</td>
<td>argv[0] = transaction id</td>
</tr>
</tbody>
</table>

**C++ PLIST and EXECOPS interactions**

The EXECOPS compiler option allows you to specify runtime options on the command line or in JCL at application invocation. NOEXECOPS indicates that runtime options cannot be so specified. When the EXECOPS compiler option is specified under MVS, Language Environment alters the MVS parameter list format by removing any runtime options present.

The PLIST compiler option indicates in what form the invoked routine should expect the argument list. You can only specify PLIST with the following value under Language Environment:

**OS**

The inbound parameter list is assumed to be in an MVS linkage format in which register 1 points to a parameter address list. No runtime options are available. Register 1 is not interrogated by Language Environment.

The EXECOPS, NOEXECOPS, and PLIST compiler options can alter the format of the argument list passed to your application, depending on the combination of options specified. The setting of EXECOPS determines whether Language Environment looks for runtime parameters in the inbound parameter list. The effects of the interactions of these options under MVS, TSO, and the various subsystems are summarized in Table 97 on page 508.
<table>
<thead>
<tr>
<th>Operating system</th>
<th>Method of invocation</th>
<th>Compiler options</th>
<th>Runtime options honored?</th>
<th>argc/argv</th>
<th>__R1/__osplst and PCBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVS</td>
<td>EXEC PGM=, PARM= &lt;runtime options&gt; / &lt;user args&gt;</td>
<td>EXECOPS (or default)</td>
<td>Yes</td>
<td>argc = number of tokenized args in &lt;user args&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>argv[0...argc-1] = tokenized args in &lt;user args&gt;</td>
<td></td>
</tr>
<tr>
<td>MVS</td>
<td>EXEC PGM=, PARM= &lt;runtime options&gt; / &lt;user args&gt;</td>
<td>NOEXECOPS</td>
<td>No</td>
<td>argc = number of tokenized args in the entire PARM string, that is, &lt;runtime options&gt; / &lt;user args&gt; argv[0...argc-1] = tokenized args in the entire PARM string</td>
<td></td>
</tr>
<tr>
<td>MVS</td>
<td>Driver link to C++ main passing noncharacter parameter list</td>
<td>PLIST(OS)</td>
<td>n/a</td>
<td>argc=1</td>
<td>Access register 1 through __osplst macro as defined in stdlib.h</td>
</tr>
<tr>
<td>TSO</td>
<td>CALL, LOADGO, execute module on TSO command line passing &lt;runtime options&gt; / &lt;user args&gt;</td>
<td>EXECOPS (or default)</td>
<td>Yes</td>
<td>argc = number of tokenized args in &lt;user args&gt;</td>
<td></td>
</tr>
<tr>
<td>TSO</td>
<td></td>
<td></td>
<td></td>
<td>argv[0...argc-1] = tokenized args in &lt;user args&gt;</td>
<td></td>
</tr>
<tr>
<td>TSO</td>
<td>CALL, LOADGO, execute module on TSO command line passing &lt;runtime options&gt; / &lt;user args&gt;</td>
<td>NOEXECOPS</td>
<td>No</td>
<td>argc = number of tokenized args in &lt;runtime options&gt; / &lt;user args&gt;</td>
<td></td>
</tr>
<tr>
<td>TSO</td>
<td></td>
<td></td>
<td></td>
<td>argv[0...argc-1] = tokenized args in &lt;user args&gt;</td>
<td></td>
</tr>
<tr>
<td>TSO</td>
<td>CALL</td>
<td>PLIST(OS)</td>
<td>n/a</td>
<td>argc=1</td>
<td>Access C PPL through __osplst as defined in stdlib.h</td>
</tr>
<tr>
<td>IMS</td>
<td>Invoke C/C++ main module</td>
<td>PLIST(OS)</td>
<td>n/a</td>
<td>argc=1</td>
<td>Access PCBs through C macros as defined in ims.h</td>
</tr>
<tr>
<td></td>
<td>Specify TARGET(IMS) also.</td>
<td></td>
<td></td>
<td>argv[0] = name of C++ main module</td>
<td></td>
</tr>
</tbody>
</table>
Table 97: Interactions of C/C++ PLIST and EXECOPS (compiler options) (continued)

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Method of invocation</th>
<th>Compiler options</th>
<th>Runtime options honored?</th>
<th>argc/argv</th>
<th>__R1/ __osplist and PCBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CICS</td>
<td>Invoke C++ main module</td>
<td>Any (or default)</td>
<td>n/a</td>
<td>argc=1 argv[0] = transaction ID</td>
<td></td>
</tr>
</tbody>
</table>

Case sensitivity under TSO

When executing under TSO with the IBM-supplied default setting of PLIST(HOST), Language Environment dynamically determines whether a command processor parameter list (CPPL) has been passed. If so, an application with a C or C++ main routine receives the TSO parameter list in an argc, argv format.

If PLIST(TSO) is in effect, the inbound parameter list is a CPPL pointed to by R1. C treats PLIST(TSO) as PLIST(HOST). A user can access the CPPL using the __osplist macro if the user specifies PLIST(OS).

Arguments passed in TSO might be case-sensitive, depending on how your C or C++ program is invoked. Table 98 on page 509 shows when the arguments are case-sensitive, based on how the C or C++ program is invoked.

Table 98: Case sensitivity of arguments under TSO

<table>
<thead>
<tr>
<th>How C or C++ program is invoked</th>
<th>Example</th>
<th>Case of argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>As TSO command</td>
<td>cprogram args</td>
<td>Mixed case (however, if you pass the arguments entirely in uppercase, the argument is lowercase)</td>
</tr>
<tr>
<td>By CALL command</td>
<td>CALL cprogram args</td>
<td>Lowercase</td>
</tr>
<tr>
<td>By CALL command with control asis</td>
<td>CALL cprogram args</td>
<td>Mixed case (however, if you pass the arguments entirely in uppercase, the argument is lowercase)</td>
</tr>
<tr>
<td>In a CLIST with control asis</td>
<td>cprogram args</td>
<td>Mixed case</td>
</tr>
<tr>
<td>As a literal passed to CLIST as a parameter</td>
<td>cprogram &amp;arg</td>
<td>Uppercase</td>
</tr>
</tbody>
</table>

Parameter passing considerations with XPLINK C and C++

C and C++ code compiled with the XPLINK option builds parameter lists using the same logical format. However, the compiler might optimize some of the parameters into registers. For more information, see z/OS Language Environment Vendor Interfaces.

COBOL parameter passing considerations

COBOL users cannot explicitly set the PLIST and EXECOPS runtime options for an enclave containing a COBOL main program. When COBOL is the main program, Language Environment sets the argument list passed to the application on initialization as follows:

- z/OS (non-CICS)
– If the COBOL main is invoked via the ATTACH SVC, a halfword-prefixed string is passed to the application after runtime options have been removed. The source of this string is dependent on the environment in which the ATTACH is issued, as follows:

- If the ATTACH is issued by z/OS to invoke a batch program, the string is specified via the EXEC statement's PARM field.
- If the ATTACH is issued by TSO to attach a Command Processor (CP), the string is specified as part of the command embedded within the CP parameter of the TSO ATTACH CP command.
- Otherwise, the string is specified via the PARM field of the ATTACH macro.

**Note:** The parameter list processing when COBOL is invoked with the ATTACH SVC can be altered with the COBOL parameter list exit IGZEPSX so that register 1 and the argument list are passed without change. If your program is not seeing the behavior mentioned previously, then see your system programmer to determine what changes were made to the COBOL parameter list exit. For information on the COBOL parameter list exit, see z/OS Language Environment Customization.

If changing IGZEPSX is not an approach that can be used in your environment, another approach is to ATTACH to a Language Environment-conforming assembler routine with MAIN=YES and PLIST=OS on the CEEENTRY macro. The Language Environment-conforming assembler routine can then invoke the COBOL program, passing the unchanged contents of register 1 (the address of the parameter list) to the COBOL program.

– If the COBOL main is not invoked by the ATTACH SVC, the halfword-prefixed string provided by the caller is passed to the application after runtime options have been removed if the following linkage is used:

- The caller of the COBOL program provides an RSA that contains a back chain (HSA) field of binary 0.
- Register 1 is nonzero.
- The word addressed by Register 1 (the first parameter pointer word) has the End of List (EOL) bit on and the parameter it addresses is aligned on a halfword or greater boundary.

– Otherwise register 1 and the argument list are passed without change.

• **TSO**

– In addition to the previous z/OS (non-CICS) considerations, if the COBOL main is invoked from a REXX clist, parameter list processing depends on the method used to invoke the COBOL program.

- If Address TSO (the default) or Address ATTCHMVS is used, the halfword-prefixed string provided by the caller is passed to the application after runtime options have been removed. Runtime options are processed. Updates made by COBOL to the parameter are not available to the calling REXX.

- If Address LINKMVS is used, the parameter list provided by the caller is passed unchanged to the application program. Runtime options, if provided are ignored. Updates made by COBOL to the parameter are available to the calling REXX.

- Address LINK, Address ATTACH, Address LINKPGM, and Address ATTCHPGM are not supported since they use a different convention for parameter lists and save area chaining.

• **z/OS UNIX**

– The parameter list consists of three parameters passed by reference:

- Argument-count: a binary fullword integer containing the number of elements in each of the arrays that is passed as the second and third parameters.

- Argument-length-list: an array of pointers. The Nth entry in the array is the address of a fullword binary integer containing the length of the Nth entry in the Argument-list (the third argument).

- Argument-list: an array of pointers. The Nth entry in the array is the address of the Nth character string passed as an argument on the spawn(), exec(), or command invocation.

• **CICS**

– If the COBOL main is invoked in a CICS environment, register 1 is passed without change.
PL/I main procedure parameter passing considerations

The format of the parameter list passed to a PL/I main procedure from the operating system is controlled by the SYSTEM compiler option and also by options on the main PROCEDURE statement.

The SYSTEM compiler option specifies the format used to pass parameters to the PL/I main procedure, and indicates the host system under which the program runs: MVS, CICS, IMS, or TSO. The SYSTEM option allows a program compiled under one system to run under another.

The NOEXECOPS procedure option indicates that runtime options are not present in the operating system parameter list. The NOEXECOPS option can be explicitly specified or implicitly defaulted. Otherwise, it is assumed that runtime options might be present in the operating system parameter list. If present, these runtime options are removed by runtime initialization before the PL/I main procedure gains control.

In order for runtime options to be passed in the operating system parameter list for SYSTEM(MVS), the PL/I main procedure must receive no parameters or receive a single parameter that is a varying character string. If this is not the case, NOEXECOPS is always defaulted.

The OPTIONS(BYVALUE) or OPTIONS(BYADDR) procedure options indicate if the main procedure parameters are passed directly or indirectly. If SYSTEM(IMS) or SYSTEM(CICS) is specified for an Enterprise PL/I for z/OS or a PL/I for MVS & VM main procedure, the OPTIONS(BYVALUE) procedure option is defaulted at compilation time, OPTIONS(BYADDR) is not permitted. When SYSTEM(CICS) and SYSTEM(IMS) is specified, Language Environment remaps the parameters to match the OPTIONS attribute BYADDR or BYVALUE of the main procedure. See “Passing arguments between routines” on page 114 for additional information about Language Environment parameter passing.

The following tables describe the interaction of the PL/I SYSTEM and NOEXECOPS options. Their effect is described in terms of the parameters that are coded on the MAIN procedure statement and also the incoming system, subsystem, or assembler parameter list as initially received by Language Environment.

<table>
<thead>
<tr>
<th>SYSTEM setting</th>
<th>No runtime options (NOEXECOPS)</th>
<th>Runtime options can be present</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM(MVS)</td>
<td>If the main procedure parameter is a single varying character string, an MVS parameter list is assumed and repackaged so the main procedure receives a halfword-prefixed string. The entire string is passed to the main procedure without change. Otherwise, the parameter list is passed without change.</td>
<td>If the main procedure parameter is a single varying character string, an MVS parameter list is assumed and repackaged so the main procedure receives a halfword-prefixed string. Any runtime options are removed from the string, and the (potentially) altered string is passed. Otherwise, the parameter list is passed without change.</td>
</tr>
<tr>
<td>SYSTEM(IMS)</td>
<td>The parameter list is passed without change.</td>
<td>Not allowed</td>
</tr>
<tr>
<td>SYSTEM(CICS)</td>
<td>The parameter list is passed without change.</td>
<td>Not allowed</td>
</tr>
<tr>
<td>SYSTEM(TSO)</td>
<td>Two levels of pointer indirection are added to the parameter list. The main procedure parameter should be a single pointer that points to the CPPL.</td>
<td>Not allowed</td>
</tr>
</tbody>
</table>

Notes:

1. NOEXECOPS is always implied for SYSTEM(CICS), SYSTEM(IMS), and SYSTEM(TSO). NOEXECOPS is also implied for SYSTEM(MVS), if the main procedure has more than one parameter or a single parameter that is not a varying character string.
2. In an IMS environment, if an assembler program is driving a PL/I transaction where LANG is not specified or LANG=NON-PLI (except Pascal) under IMS V4R1, the parameter passes through without change. Otherwise, one level of indirection is removed from the parameter.

If an assembler program is driving a transaction program written in Enterprise PL/I for z/OS or PL/I for MVS & VM, the main procedure of the transaction must be compiled with SYSTEM(MVS) option; the main procedure receives the parameter list passed from the assembler program in MVS style.
Appendix E. Object library utility

The object library utility is used to update libraries of object modules. A library is a partitioned data set (PDS or PDSE) with object modules as members.

Object libraries provide for convenient packaging of object modules. With the Object Library Utility, a library can contain object modules with L-names, object modules with S-names, and object modules with writable static data. The Object Library Utility is used to create information, such as which members contain defined L-names, S-names, or writable static data. This information is stored in a special member of the library that will be referred to as the Object Library Utility directory.

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

Creating an object library

You can create an object library under batch or TSO.

Under batch

Under MVS batch, the following cataloged procedures include an Object Library Utility step:

**EDCLIB**
Maintain an object library.

**EDCCLIB**
Compile and maintain an object library.

For more information about the data sets used with the Object Library Utility, see z/OS XL C/C++ User's Guide.

You can specify options for the Object Library Utility step that generate a library directory, add or delete members of a directory, or generate a map of library members and defined external symbols. This topic shows you how to specify these options under MVS batch.

To compile the C program `WALTER.SOURCE(SUB1)` for L-names and add to `WALTER.SOURCE.OBJ(SUB1)`, use the following JCL. The Object Library Utility directory for the library, `WALTER.SOURCE.OBJ`, is updated in the process.

```
//COMPILE EXEC EDCCLIB,INFILE='WALTER.SOURCE(SUB1)',CPARM='LO',
//      LIBRARY='WALTER.SOURCE.OBJ',MEMBER='SUB1'
```

To request a map for the library `WALTER.SOURCE.OBJ`, use:

```
//OBJLIB EXEC EDCLIB,OPARM='MAP',LIBRARY='WALTER.SOURCE.OBJ'
```

The following example creates a new Object Library Utility directory. If the directory already exists, it is updated:

```
//DIRDIR   EXEC EDCLIB,
//         LIBRARY='LUCKY13.CXX.OBJMATH',
//         OPARM='DIR'
```

To create a map:

```
//MAPDIR   EXEC EDCLIB,
//         LIBRARY='LUCKY13.CXX.OBJMATH',
//         OPARM='MAP'
```
To add new members to an object library, use the ADD option to update the directory. For example, to add a new member named MA191, code:

```sql
//ADDDIR EXEC EDCLIB,
//       LIBRARY='LUCKY13.CXX.OBJMATH',
//       OPARM='ADD MA191',
//       OBJECT='DSNAME=LUCKY13.CXX.OBJ(OBJ191),DISP=SHR'
```

To delete a member from an object library, use the DEL option to keep the directory up-to-date. For example, to delete a member named OLDMEM, code:

```sql
//DELDIR EXEC EDCLIB,
//       LIBRARY='LUCKY13.CXX.OBJMATH',
//       OPARM='DEL OLDMEM'
```

**Under TSO**

The Object Library Utility has the following syntax:

```
C370LIB ADD (libname(membername))
C370LIB DEL (libname(membername))
C370LIB MAP (libname)
C370LIB LIST (map)
C370LIB DIR (libname)
```

**ADD**

Adds (or replaces) an object module in an object library.

If the ADD function is used to insert an object module in a member of a library that already exists, the previous member is deleted before the insert unless the source data set is the same as the target data set, in which case the member is not deleted and only the Object Library Utility directory is updated as appropriate.

**DEL**

Deletes an object module from an object library.

**MAP**

Lists the names (entry points) of object library members.

**DIR**

Builds the Object Library Utility directory member. The Object Library Utility directory contains the names (entry points) of library members.

The DIR function is only necessary if object modules were previously added or deleted from the library without using C370LIB.

**LIB (libname(membername))**

Specifies the target data set for the ADD and DEL functions. The data set name must contain a member specification to indicate which member is to be created, replaced, or deleted.

**OBJ (objname)**

Specifies the source data set containing the object module that is to be added to the library. If you do not specify a data set name, the target data set specified in LIB(libname(membername)) is used as the source.

**LIB (libname)**

Specifies the object library for which a map is to be produced or for which a Object Library Utility directory is to be built.
LIST (map)

Specifies the data set that is to contain the library map. If an asterisk (*) is specified, the library map is directed to your terminal. If you do not specify a data set name, a name is automatically generated using the library name and the qualifier MAP. If the input library data set is called TEST.OBJ and your user prefix is FRANK, the data set name generated for the map is FRANK.TEST.OBJ.MAP.

Under TSO, you can use either the C370LIB CLIST or the CC CLIST using the parameter C370LIB. The C370LIB parameter of CC CLIST specifies that if the object module from the compile is directed to a member of a PDS, then the Object Library Utility directory is to be updated. This step is the equivalent to a compile and C370LIB ADD step. If the C370LIB parameter is specified and the object module is not directed to a member of a PDS, the C370LIB parameter is ignored.

Object library utility map

The Object Library Utility produces a listing for a given library when the MAP command is specified. The listing contains information on each member of the library. A representative example is shown in Figure 115 on page 515.

---

<table>
<thead>
<tr>
<th>Object Library Utility Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>C370LIB:5647A01 V2 R9 M00 IBM Language Environment 1999/12/16 16:22:43</td>
</tr>
</tbody>
</table>

- * Member Name: ASMSTUFF (D) 1998/04/22 11:46:39 *
  569623400 R01 M01 *

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

- * Member Name: CSTUFF (D) 1998/04/22 11:46:39 *
  5688216 R32 M00 *

- (L) Function Name: foo
  (WL) External Name: this_int_is_in_writable_static_and_its_name_will_wrap_because_it_is_too_long

- * Member Name: CXSTUFF (D) 1998/04/22 11:46:39 *
  5688216 R32 M00 *

- User Comment: This is a user comment in CXSTUFF

---

Figure 115: Object library utility map

Map Heading

The heading contains the product number, the compiler release number, the compiler version number, and the date and time the Object Library Utility step commenced. The name of the library immediately follows the heading. To the right of the name of the library is the start time of the last Object Library Utility step that updated the Object Library Utility directory.
**Member Heading**

The name of the object module member is immediately followed by the ID of the processor that produced the object module. The processor ID is based on the presence of an END record in the object module having the processor information in the appropriate format. If this information is not present, the Processor ID field is not listed.

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

- **(P)** Indicates that the timestamp is extracted from the object module from the date form of `#pragma comment` or from the timestamp form of `#pragma comment`, whichever comes first.
- **(D)** Indicates that the timestamp is based on the time that the Object Library Utility DIR command was last issued.
- **(T)** Indicates that the timestamp is the time that the ADD command was issued for the member.

**User Comments**

The user form of comments generated by `#pragma comment` is displayed. These comments are extracted from the END record. It is possible to manually add such comments on multiple END records and have them displayed in the listing. See z/OS XL C/C++ Language Reference for more information on the END record.

**Symbol Information**

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

- **'L'** Indicates that the name is an L-name.
- **'S'** Indicates that the name is an S-name.
- **'W'** Indicates that this is a writable static object. If no 'W' is present, then this is not a writable static object.
- **'WL'** Indicates that this is both an L-name and in writable static.
Appendix F. Using the systems programming environment

Note: This topic applies to C applications only.

As a C routine executes, facilities from the Language Environment common library are invoked to set up the execution environment in order to handle termination activities and provide storage management, error handling, runtime options parsing, ILC, and debugging support. In addition, the C library functions are in the Language Environment common library.

For situations in which not all of these services are needed, the system programming facilities of C can provide a limited environment.

System programming facilities allow you to run applications without using the Language Environment common library, or with just the C library functions, and to:

- Develop C applications that do not require the Language Environment common library on the machines on which they run.
- Use C as an assembler language substitute to, for example, write exit routines for MVS, TSO, or JES.
- Develop applications featuring:
  - A persistent C environment, in which a C environment is created once and used repeatedly for C function execution from any language.
  - Co-routines that use a two-stack model, as in client-server style applications. In this style, the user application calls on the applications server to perform services independently of the user and then return to the user.

For more information on the system programming facilities of C, see z/OS XL C/C++ Programming Guide.

This topic discusses how to build these applications once you have compiled them with the C compiler. You must compile these programs with the NOSTART option.

Building freestanding applications

Freestanding applications need to be linked with specific alternate initialization routines.

To explicitly include an alternative initialization routine under MVS, use the linkage editor INCLUDE and ENTRY control statements. To include the alternate initialization routines, you must allocate CEE.SCEESPC to the SYSLIB DD. For example, the following linkage editor control stream might be used to specify EDCXSTRT as an alternate initialization routine (another example is shown in Figure 118 on page 518):

```
INCLUDE SYSLIB(EDCXSTRT)
ENTRY EDCXSTRT
INCLUDE SYSLIB
```

*Figure 116: Specifying alternate initialization at link-edit time*

If you are building freestanding applications under MVS, CEE.SCEESPC must be included in the link-edit SYSLIB concatenation. Also, if C library functions are needed, CEE.SCEESPC must precede CEE.SCEELKED.

The routines to support this function (EDCXSTRT and EDCXSTRL) are CEESTART replacements in your module. You must specify NOSTART compiler option when compiling the file that contains the main function. Therefore, the appropriate EDCXSTRn routine must be explicitly included at link-edit.
A simple freestanding routine that requires a C library function is shown in Figure 117 on page 518.

```c
#include <stdio.h>
main() {
    puts("Hello, World");
    return 3999;
}
```

*Figure 117: Simple freestanding routine*

This routine, RET3999, is compiled with nostart compiler option and link-edited using control statements in Figure 118 on page 518. It is assumed that:

- The object module is available to the linkage-editor by using an OBJECT DD statement.
- CEE.SCEESPC and CEE.SCEELKED libraries are specified on a SYSLIB DD statement.
- The intended load module member name is specified on a SYSLMOD DD statement.

The CEE.SCEERUN runtime load library must be available at runtime because it contains the C library function `puts()`.

```
#include SYSLIB(EDCXSTRL)
INCLUDE OBJECT
ENTRY EDCXSTRL
```

*Figure 118: Link-edit control statements used to build a freestanding MVS routine*

Figure 119 on page 518 shows how to compile and link a freestanding program by using the cataloged procedure EDCCL. See *z/OS XL C/C++ Programming Guide* for more information about EDCCL.

```
//* Appropriate JOB card
//*-----------------------------------------------------------------
//**************************************************
//*** COMPILE AND LINK USING EDCXSTRL AS ENTRY POINT
//**************************************************
//C106001     EXEC    EDCCL,
//      INFILE='ANDREW.SPC.SOURCE(C106000)',
//      OUTFILE='ANDREW.SPC.LOAD(C106000),DISP=SHR',
//      CPARM='OPT(2),NOSEQ,NOMAR,NOSTART',
//      LPARM='RMODE=ANY,AMODE=31'
//LKED.SYSLIB DD DSN=CEE.SCEESPC,DISP=SHR
//            DD DSN=CEE.SCEELKED,DISP=SHR
//LKED.SYSIN  DD *
INCLUDE SYSLIB(EDCXSTRL)
ENTRY EDCXSTRL
/*
```

*Figure 119: Compile and link by using the cataloged procedure EDCCL*

**Special considerations for reentrant modules**

A simple freestanding routine that does not require C library functions is shown in Figure 120 on page 519. This routine uses the `exit()` function, which is normally part of the C library but (like `sprintf()`)
is available to freestanding routines without requiring the dynamic library. This routine is not naturally reentrant, but the resulting load module is reentrant.

```c
#include <stdlib.h>

int main() {
    static int i[5]={0,1,2,3,4};
    exit(320+i[1]);
}
```

*Figure 120: Sample reentrant freestanding routine*

The JCL required to build and execute this routine is shown in *Figure 121 on page 519*. The bracketed numbers in the figure refer to the comments that follow.

```jcl
//PRLK      EXEC   PGM=EDCPRLK,PARM='MAP,NCAL' [Figure 121 on page 519-1]
//STEPLIB   DD     DSN=CEE.SCEERUN,DISP=SHR
//SYSMSG5   DD     DSN=CEE.SCEEMSGP(EDCPMSG5),DISP=SHR
//SYSOUT    DD     SYSDOUT=* [Figure 121 on page 519-2]
//SYSMOD    DD     DSN=&GOBJ,SPACE=(TRK,(1,1)),UNIT=SYSDA,
//           DCB=(BLKSIZE=400,RECFM=FB,LRECL=80),
//           DISP=(MOD,PASS)
//SYSIN     DD     DSN=RETS321.OBJ,DISP=SHR [Figure 121 on page 519-3]
//*
//LKED      EXEC   PGM=HEWL,PARM='MAP,XREF,LIST' [Figure 121 on page 519-4]
//SYSUT1    DD     SPACE=(CYL,1),UNIT=SYSDA
//PRELINK   DD     DSN=&GOBJ,DISP=(OLD,DELETE)
//SYSLIN    DD     + INCLUDE SYSLIB(EDCXSTRT) [Figure 121 on page 519-5]
//INCLUDE PRELINK
//INCLUDE SYSLIB(EDCXEXIT) [Figure 121 on page 519-6]
//INCLUDE SYSLIB(EDCRCINT) [Figure 121 on page 519-7]
//*
//SYSPRINT  DD     SYSDOUT=*
//SYSMOD    DD     DSN=&GOSET(GO),
//UNIT=SYSDA,SPACE=(TRK,(1,1,1)),
//DISP=(NEW,PASS)
//SYSLIB    DD     DSN=CEE.SCEESPC,DISP=(SHR,PASS)
//GO        EXEC   PGM=*.LKED.SYSLMOD
```

*Figure 121: Building and running a reentrant freestanding MVS routine*

**Notes**

*Figure 121 on page 519-1*  
The prelinker must be used for modules compiled with the RENT compiler option.

*Figure 121 on page 519-2*  
This is the object module created by compiling the sample module with the RENT and NOSTART compiler options.

*Figure 121 on page 519-3*  
The output from the prelinker is made available to the linkage editor.

*Figure 121 on page 519-4*  
The alternate initialization routine (EDCXSTRT in this example) must be explicitly included in the module. If this is not the first CSECT in the module it must be explicitly named as the module entry point.

*Figure 121 on page 519-5*  
The prelinked output is included in the load module.
Building system exit routines

There are no special considerations for building system exit routines. These routines can be linked with their callers or dynamically loaded and invoked. CEE.SCEESPC must be available at link-edit. If C library functions are required by the exit routines, the CEE.SCEELKED library must also be made available after CEE.SCEESPC. If the routines were compiled with OPT(2), the entry point must be explicitly named in the link-edit input.

**Note:** You must compile these programs with the NOSTART option.

Building persistent C environments

There are no special considerations for building applications that use persistent C environments. The data set CEE.SCEESPC contains the object modules to be included.

If C library functions are required by any routine called in this environment, the library stub routines should also be made available at link time after CEE.SCEESPC.

**Note:** You must compile these programs with the NOSTART option.

Building user-server environments

To build your server application, follow the rules for building a freestanding application as described in “Building freestanding applications” on page 517.

There are no special considerations for building user applications. The automatic call facility causes the right routines from SYSLIB to be included.

**Note:** You must compile servers with the NOSTART option.

Summary of types

<table>
<thead>
<tr>
<th>Type of application</th>
<th>How it is called</th>
<th>Module entry point</th>
<th>Data sets required at execution time</th>
<th>Runtime options and other considerations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A mainline function that requires no C-specific library functions.</td>
<td>From the command line, JCL, or an EXEC or CLIST.</td>
<td>EDCXSTRT must be explicitly included at bind time.</td>
<td>None.</td>
<td>Runtime options are specified by #pragma runopts in the compilation unit for the main() function. The HEAP and STACK options are honored. STACK defaults to above the 16M line.</td>
</tr>
</tbody>
</table>

Table 100: Summary of types
Table 100: Summary of types (continued)

<table>
<thead>
<tr>
<th>Type of application</th>
<th>How it is called</th>
<th>Module entry point</th>
<th>Data sets required at execution time</th>
<th>Runtime options and other considerations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A mainline function that requires C library functions.</td>
<td>From the command line, JCL, or an EXEC or CLIST.</td>
<td>EDCXSTR must be explicitly included at bind time.</td>
<td>C library functions.</td>
<td>Runtime options are specified by <code>#pragma runopts</code> in the compilation unit for the <code>main()</code> function. The TRAP, HEAP and STACK options are honored, but the stack defaults to above the 16M line.</td>
</tr>
<tr>
<td>A mainline function that uses storage pre-allocated by the caller.</td>
<td>From Assembler code.</td>
<td>C library functions are optional; the caller must load these functions and pass their addresses to EDCXSTRX, if required to by the application.</td>
<td>Runtime options are specified by <code>#pragma runopts</code> in the <code>main()</code> function. The TRAP option is honored if C library functions are required.</td>
<td></td>
</tr>
<tr>
<td>An exit.</td>
<td>Typically from assembler code, with a structured parameter list.</td>
<td>C library functions, if required.</td>
<td>Runtime options are specified by <code>#pragma runopts</code> in the compile unit for the entry point. The HEAP and STACK options are honored, but the stack defaults to be above the 16M line. The TRAP option is honored if C library functions are required.</td>
<td></td>
</tr>
<tr>
<td>Type of application</td>
<td>How it is called</td>
<td>Module entry point</td>
<td>Data sets required at execution time</td>
<td>Runtime options and other considerations</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------</td>
<td>--------------------</td>
<td>--------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>A C subroutine called from Assembler language using a pre-established persistent environment.</td>
<td>A handle, the address of the subroutine, and a parameter list are passed to EDCXHOTU.</td>
<td></td>
<td>C library functions are optional, depending on the way the handle was set up.</td>
<td>Runtime options are specified by #pragma runopts in any compile unit. The HEAP and STACK options are honored, but the stack defaults to above the 16M line. The TRAP option is honored if C library functions are called for. The runopts in the first object module in the link-edit that contains runopts prevails, even if this compilation unit is part of the calling application. The environment is established by calling EDCXHOTC or EDCXHOTL (if library functions are required). These functions return a value (the handle), which is used to call functions that use the environment.</td>
</tr>
<tr>
<td>A server.</td>
<td>User code includes a stub routine that calls EDCXSRVI. This causes the server to be loaded and control to be passed to its entry point.</td>
<td>EDCXSTRT or EDCXSTRL, depending on whether the server needs C library functions.</td>
<td>C library functions, if required by the server code.</td>
<td>Runtime options are the same as for EDCXSTRL or EDCXSTRT. The author of the server must supply stub routines that call EDCXSRVI and EDCXSRVN to initialize and communicate with the server. These are bound with the user application.</td>
</tr>
<tr>
<td>A user of an application server.</td>
<td></td>
<td></td>
<td>The server and C library functions, if required by the server.</td>
<td>The author of the server must supply stub routines which call EDCXSRVI and EDCXSRVN to initialize and communicate with the server.</td>
</tr>
</tbody>
</table>
Appendix G. Sort and merge considerations

This topic discusses the runtime aspects of sort and merge operations. For details on the compile-time aspects of sort and merge, including instructions on coding the sort and merge procedures, see your compiler programming guide.

Under Language Environment, you can invoke the sort facility to sort or merge records in a particular sequence. A sort operation takes an unordered sequence of input data, arranges it according to a specified key or pattern, and places it into an output file. A merge operation compares two or more files that have already been sorted according to an identical key and combines them in a specified order in an output file.

To invoke the sort facility in Language Environment, you can use either of the following:

- An HLL construct
  - COBOL's SORT and MERGE statements. (The SORT and MERGE statements are not supported when running under z/OS UNIX.)
  - PL/I's PLISRTx interface, where x is replaced by A, B, C, or D
    You cannot call the PLISRTx interface under CICS.
- A method other than an HLL construct (for example, assembler routines, JCL, or ISPF).

Under Language Environment, your IBM sort/merge licensed program must be DFSORT or an equivalent that honors the DFSORT extended parameter list. Whenever DFSORT is mentioned in this topic, you can use any equivalent SORT product.

**Restriction:** SORT and MERGE is not supported in a POSIX(ON) environment.

**Invoking DFSORT directly**

For information about using the methods to run DFSORT directly with JCL or to invoke DFSORT directly from an assembler program, see [z/OS DFSORT Application Programming Guide](https://www.ibm.com/support/docview.wss?uid=swg27036733). Also see that topic for details on DFSORT’s many built-in features that you can use to eliminate the need for writing program logic (for example, the INCLUDE, OMIT, OUTREC, and SUM statements).

**Using the COBOL SORT and MERGE verbs**

This topic contains a high-level overview of COBOL SORT and MERGE verbs. It is designed to introduce you to concepts that help you understand some of the special considerations for using these COBOL statements in Language Environment. For a detailed description of how to use SORT and MERGE, see the appropriate version of the COBOL programming guide in the COBOL library at Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733).

A COBOL program that contains a sort operation can be organized so that an input procedure reads and operates on one or more input files before the files are actually sorted. To specify the input procedure:

```
SORT...INPUT PROCEDURE
```

You can also specify an output procedure that processes the files after they are sorted:

```
SORT...OUTPUT PROCEDURE
```

These input and output procedures can be used to add, delete, alter, edit, or otherwise modify the records.
Sort and merge considerations

You can also sort records under COBOL without any processing by the input and output procedures. For example, to read records into a new file for sorting without any preliminary processing, specify:

```
SORT...USING
```

To transfer sorted records to a file without any further processing, specify:

```
SORT...GIVING
```

User exit considerations

SORT or MERGE COBOL verbs can trigger COBOL-generated user exits (E15 for sort, E35 for merge). These exits include any input or output procedures. However, the exits are not triggered when a COBOL USING or GIVING statement is in effect and the files qualify for FASTSRT.

Language Environment treats the COBOL-generated exits differently than those requested by a direct invocation of DFSORT. Language Environment treats user exits triggered by COBOL SORT or MERGE as part of the enclave of the routine that invoked DFSORT; the SVC LINK used to invoke DFSORT is not considered by Language Environment to initiate a new implicit nested enclave. This is not the case for direct invocations of DFSORT, which do result in the creation of a new nested enclave. See Chapter 31, “Using nested enclaves,” on page 469 for more information on nested enclaves, and see z/OS DFSORT Application Programming Guide for more information on direct invocations of DFSORT.

Condition handling considerations

This topic summarizes how Language Environment condition handling behaves when a Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, or VS COBOL II routine is involved in a SORT or MERGE operation.

Program interrupts

User handlers established by the routine that initiated the SORT/MERGE are able to handle program interrupts as they are presented to the condition manager by a condition token. Normal condition handling as described in Chapter 15, “Introduction to Language Environment condition handling,” on page 167 occurs.

Establishment of HLL-specific handlers and user handlers is not supported while in a SORT input or output procedure. The results are unpredictable, and the condition handler does not attempt to diagnose this case.

HLL-specific handlers and user handlers established by a routine called by an input or output procedure are able to handle program interrupts. However, because these exits are typically invoked many times (equivalent to the number of records being sorted for each exit), it is recommended that you register the handler within the application that initiated the SORT/MERGE in order to avoid overhead.

Language Environment-signaled conditions

HLL-specific handlers and user handlers established by the routine that initiated the SORT/MERGE are able to handle any condition signaled by Language Environment. Normal condition handling as described in Chapter 15, “Introduction to Language Environment condition handling,” on page 167 occurs.

Abends

When there is an abend, the DFSORT ESTAE exit intercepts the abend, and performs various cleanups and recoveries. Informational dumps and messages are produced as appropriate. The abend is then percolated and eventually intercepted by the Language Environment ESTAE exit. Condition handling then continues as described in Chapter 15, “Introduction to Language Environment condition handling,” on page 167.

By the time the Language Environment ESTAE exit intercepts the abend, the SORT has been terminated. Language Environment moves the current resume cursor to the return point where SORT was invoked and
reflects the deletion of stack frames (and associated load modules) following the SORT invocation. Any user condition handlers associated with these stack frames (those following the SORT invocation) do not get control.

### Using the PL/I PLISRTx interface

This topic contains a high-level overview of the PLISRTx interfaces to DFSORT. It is designed to introduce you to concepts that help you understand some of the special considerations for using these PL/I interfaces in Language Environment. For a detailed description of how to use PLISRTx, see the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

PL/I provides a SORT interface called PLISRTx. When you make a call to PLISRTx, you replace x with A, B, C, or D, depending on whether your input comes from a data set or a PL/I subroutine, and whether your output is to be written to a data set or processed by a PL/I subroutine:

- **PLISRTA**: Unsorted input is read from a data set and then sorted. The sorted output is written to a data set.
- **PLISRTB**: Unsorted input is provided and processed by a PL/I subroutine before sorting. The sorted output is written to a data set.
- **PLISRTC**: Unsorted input is read from a data set and then sorted. The sorted output is then processed by a PL/I subroutine.
- **PLISRTD**: Unsorted input is provided and processed by a PL/I subroutine before sorting. The sorted output is then processed by a PL/I subroutine.

In the call to PLISRTx, you also pass information about your data, using the SORT and RECORD arguments, and specify the maximum amount of storage you will allow DFSORT to use.

**User exit considerations**

Your input handling subroutine and output handling subroutine must be written in PL/I. PL/I generates a DFSORT E15 exit for your input handling subroutine and a DFSORT E35 exit for your output handling subroutine.

A call to one of the PLISRTx interfaces might trigger a call to user exit E15, E35, or both, depending on whether a subroutine is to process your input before sorting, or your output after sorting, as shown in Table 101 on page 525.

<table>
<thead>
<tr>
<th>PL/I sort interface</th>
<th>DFSORT exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLISRTA</td>
<td>None</td>
</tr>
<tr>
<td>PLISRTB</td>
<td>E15</td>
</tr>
<tr>
<td>PLISRTC</td>
<td>E35</td>
</tr>
<tr>
<td>PLISRTD</td>
<td>E15 and E35</td>
</tr>
</tbody>
</table>

Language Environment treats the generated E15 and E35 exits differently than those requested by a direct invocation of DFSORT. Language Environment treats user exits triggered by PLISRTx as part of the enclave of the routine that invoked DFSORT; the SVC LINK used to invoke DFSORT is not considered by Language Environment to initiate a new implicit nested enclave. See Chapter 31, “Using nested enclaves,” on page 469 for more information on nested enclaves, and see z/OS DFSORT Application Programming Guide for more information on direct invocations of DFSORT.
Condition handling considerations

Input and output handling subroutines can issue GOTOs. If you need to deactivate the SORT program for any reason while in one of these exits, issue a GOTO out of the subroutine.

Program interrupts and Language Environment-signaled conditions

PL/I ON-units can be established in any of the following:

- The routine that made a call to PLISRTx.
- The input(E15) or output(E35) procedure.
- A routine called by the input or output procedure.

These ON-units can handle program interrupts and Language Environment-signaled conditions. Normal condition handling, as described in Chapter 15, “Introduction to Language Environment condition handling,” on page 167, occurs.

Abends

ON-units do not have the opportunity to handle abends that arise during a sort operation.

When there is an abend, the DFSORT ESTAE exit intercepts the abend and performs various clean-ups and recoveries. Informational dumps and messages are produced as appropriate. The abend is then percolated and eventually the Language Environment ESTAE exit intercepts it. Condition handling then continues as described in Chapter 15, “Introduction to Language Environment condition handling,” on page 167.

By the time the Language Environment ESTAE exit intercepts the abend, the SORT has been terminated. Language Environment moves the current resume cursor to the return point where SORT was invoked and reflects the deletion of stack frames (and associated load modules) after the SORT invocation. Any user condition handlers associated with these stack frames (those following the SORT invocation) do not get control.

When running DFSORT (or an OEM SORT function), it is recommended that the TRAP(ON), or TRAP(ON,SPIE) Language Environment runtime option be specified. This will ensure that the Language Environment ESPIE is available to process expected internal Language Environment program interrupts.
Appendix H. Running COBOL programs under ISPF

This topic applies to COBOL users only.

When you code your application using ISPF panels, you can gain interactive access to your COBOL application.

1. If you attempt to pass runtime options to a COBOL program that is invoked from ISPF, the runtime options will be treated as program arguments.
2. Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM and COBOL/370 programs are allowed to run concurrently in both screens of the ISPF split screen mode.

⚠️ CAUTION: Prior versions of COBOL may not run concurrently in both screens of the ISPF split screen mode.
Appendix I. Language Environment macros

The macros identified in this topic are provided as programming interfaces for customers by Language Environment.

Attention:
Do not use as programming interfaces any Language Environment macros other than those identified in this topic.

All macros listed here are provided as General-Use Programming Interfaces.

- **CEECAA** (see “CEECAA macro — Generate a CAA mapping” on page 404)
- **CEEDSA** (see “CEEDSA macro — Generate a DSA mapping” on page 404)
- **CEEENTRY** (see “CEEENTRY macro— Generate a Language-Environment-conforming prolog” on page 400)
- **CEEFETCH** (see “CEEFETCH macro — Dynamically load a routine” on page 409)
- **CEELOAD** (see “CEELOAD macro — Dynamically load a Language Environment-conforming routine” on page 407)
- **CEEPPA** (see “CEEPPA macro — Generate a PPA” on page 404)
- **CEERELES** (see “CEERELES macro — Dynamically delete a routine” on page 416)
- **CEETERM** (see “CEETERM macro — Terminate a Language Environment-conforming routine” on page 403)
- **CEEXOPT** (see Chapter 9, “Using runtime options,” on page 101)
- **CEEXPIT** (see “Macros that generate the PreInit table” on page 431)
- **CEEXPITY** (see “Macros that generate the PreInit table” on page 431)
- **CEEXPITS** (see “Macros that generate the PreInit table” on page 431)
- **__csplist** (see “C and C++ parameter passing considerations” on page 503)
- **__osplist** (see “C and C++ parameter passing considerations” on page 503)
- **__pcblist** (see “C and C++ parameter passing considerations” on page 503)
- **__R1** (see “C and C++ parameter passing considerations” on page 503)
Language Environment macros
Appendix J. PL/I macros that activate variables

Several PL/I macros shipped with Language Environment activate (%ACT) variables on behalf of the user program. Using them, code developers can use common Language Environment data types. Use of these variable names by a user program will result in a compile error. Table 102 on page 531 lists the macros and the preprocessor variable names.

These PL/I preprocessor variables correspond with the Language Environment data types of the same name. Use them much as you would standard PL/I attributes. They can be used in combination with:

- Storage class attributes (such as BASED)
- Scope attributes (such as EXTERNAL)
- Alignment attributes (such as ALIGNED)
- Aggregation attributes, including dimensions and structure level numbers; in fact, some of the data types require the use of level numbers

The Language Environment data types must not be used in combination with:

- Arithmetic attributes, including BASE, SCALR, MODE, PRECISION and PICTURE
- String attributes, including BIT, CHAR, GRAPHIC, VARYING and PICTURE
- Program control data Attributes, including AREA, ENTRY, FILE, LABEL, OFFSET, POINTER, TASK and VARIABLE
- The LIKE attribute, although you can LIKEN another identifier to one declared using the Language Environment data types

It is strongly recommended that you always code the Language Environment data type as the last attribute in any identifier’s declaration. Unlike true PL/I attributes, order sometimes counts.

There is another difference between these Language Environment data types and true PL/I attributes: some of them may not be used as parameter descriptors (in the parameter list of the ENTRY attribute). Consequently, some of the data types are available in two forms: with and without a "_PARM" suffix. In these cases, you must use the _PARM version when you specify a parameter descriptor, and the other version in all other contexts.

One final difference between the Language Environment data types in the following tablet and true PL/I attributes which you must be aware of is that the Language Environment data types must be treated as reserved words. Whereas PL/I attribute names can also be used as identifier names, the Language Environment data type names may not be used in any context other than that of an attribute.

Table 102: Variables activated by PL/I macros

<table>
<thead>
<tr>
<th>PL/I macro</th>
<th>Variables activated</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEIBMAW</td>
<td>INT2</td>
</tr>
<tr>
<td></td>
<td>INT4</td>
</tr>
<tr>
<td></td>
<td>FLOAT4</td>
</tr>
<tr>
<td></td>
<td>FLOAT8</td>
</tr>
<tr>
<td></td>
<td>FLOAT16</td>
</tr>
<tr>
<td></td>
<td>COMPLEX4</td>
</tr>
<tr>
<td></td>
<td>COMPLEX8</td>
</tr>
<tr>
<td></td>
<td>COMPLEX16</td>
</tr>
<tr>
<td></td>
<td>VSTRING</td>
</tr>
<tr>
<td>PL/I macro</td>
<td>Variables activated</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>VSTRING_PARM</td>
<td></td>
</tr>
<tr>
<td>CHAR80</td>
<td></td>
</tr>
<tr>
<td>FEEDBACK</td>
<td></td>
</tr>
<tr>
<td>FEED_BACK</td>
<td></td>
</tr>
<tr>
<td>FEEDBACK_PARM</td>
<td></td>
</tr>
<tr>
<td>FEED_BACK_PARM</td>
<td></td>
</tr>
<tr>
<td>CEE_ENTRY</td>
<td></td>
</tr>
<tr>
<td>CEE_ENTRY_PARM</td>
<td></td>
</tr>
<tr>
<td>CEEIBMCT</td>
<td>FBCHECK</td>
</tr>
</tbody>
</table>
Appendix K. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed email message to mhvrdfs@us.ibm.com.

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- z/OS TSO/E Primer
- z/OS TSO/E User's Guide
- z/OS ISPF User's Guide Vol I

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with 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 are considered 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 the 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, 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 is placed 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 to provide 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, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

**? indicates an optional syntax element**
The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark 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 the 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.

**! indicates a default syntax element**
The exclamation mark (!) symbol indicates 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 dotted decimal number can specify the ! 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 the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the 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 applies only 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.

*** indicates an optional syntax element that is repeatable**
The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero 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, 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 can write HOST STATE, but you cannot write HOST HOST.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

**+ indicates a syntax element that must be included**

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the 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 repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.
This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for the Knowledge Centers. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you 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
Site Counsel
2455 South Road
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

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 measurements 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 or withdrawal 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 illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or
reproduce, distribute or display these publications or any portion thereof outside your enterprise, without
the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either
express or implied, to the publications or any information, data, software or other intellectual property
contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of
the publications is detrimental to its interest or, as determined by IBM, the above instructions are not
being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable
laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE
PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED,
INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT,
AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies
or other technologies to collect product usage information, to help improve the end user experience, to
tailor interactions with the end user, or for other purposes. In many cases no personally identifiable
information is collected by the Software Offerings. Some of our Software Offerings can help enable you to
collect personally identifiable information. If this Software Offering uses cookies to collect personally
identifiable information, specific information about this offering’s use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect
each user’s name, email address, phone number, or other personally identifiable information for purposes
of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling
them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect
personally identifiable information from end users via cookies and other technologies, you should seek
your own legal advice about any laws applicable to such data collection, including any requirements for
notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see
IBM’s Privacy Policy at ibm.com/privacy and IBM’s Online Privacy Statement at ibm.com/privacy/details in
the section entitled “Cookies, Web Beacons and Other Technologies,” and the “IBM Software Products
and Software-as-a-Service Privacy Statement” at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, 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.
Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

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 (www.ibm.com/legal/copytrade.shtml).

UNIX is a registered trademark of The Open Group in the United States and other countries.
Index

Special Characters
__csplist macro 504
__osplist macro 504
__pcblist macro 504
__R1 macro 504
CEE_RUNOPTS environment variable
  specifying runtime options at invocation 102
, (comma) 104
' (apostrophe) 107
@DELETE service routine for preinitialization
  components of 461
    return/reason codes for 461
@EXCEPRTN service routine for preinitialization
  components of 462
    return/reason codes for 463, 464
@FREESTORE service routine for preinitialization
  components of 462
    return/reason codes for 462
@GETSTORE service routine for preinitialization
  components of 461
    return/reason codes for 462
@LOAD service routine for preinitialization
  components of 460
    return/reason codes for 460
@MSGRTN service routine for preinitialization
  components of 465
    return/reason codes for 466
/ (slash)
  specifying in PARM parameter 67
& (ampersand) 107
= (equal) 67

Numerics
16M line
  COBOL programs that run above must be reentrant 121
A
abend codes
  abend 4093, reason code 60 511
  CEEAUE_RETC field of CEEBXITA and ABTERMENC(ABEND) 136
    in CICS 361
    short-on-storage condition and 358
ABEND command
  CEE3ABD callable service and 424
  CEESGL callable service and 424
    table of equivalent Language Environment services 424
abends
  ABEND command and 424
  CEEPIPI interface to preinitialization 447
  CICS
    assembler user exit and EXEC CICS ABEND 382
    EXEC CICS HANDLE ABEND and 360
    forcing database rollback 362
abends (continued)
  CICS (continued)
    nested conditions and 471
    short-on-storage condition and 358
    dump, requesting in CEEBXITA assembler user exit 382
    IMS, creating system abends under 369
    Language Environment-generated 171
    nested conditions and 207
    nested enclaves and
      created by C system() 475
      created by EXEC CICS LINK or EXEC CICS XCTL 471
      created by SVC LINK 472
    percolating
      in CEEBXITA 171, 378
      methods of percolating 171
    q_data_token and 244
    short-on-storage condition can cause 358
    sort and merge operations, occurring in
      in COBOL 524
      in PL/I 526
      specifying in CEEBXITA 378
      terminating with an abend 135
    TRAP runtime option and CEEBXITA 378
abort() function
  C condition handling semantics and 186
  HLL user exit and 388
    in a preinitialized environment 434
  SIGABRT and 184
ABTERMENC runtime option
  using to create system abends under MVS 369
  using to terminate with abend code or return and reason
  codes 135
ACCEPT statement 276
accessibility
  contact IBM 533
  features 533
add_entry
  return codes from 453
  syntax description 452
additional heap
  tuning the heap 153
AFHWL cataloged procedure 95
AFHWLG cataloged procedure 96
AFHWN cataloged procedure 96
AFHWRLK — Fortran library replacement tool 14
AIB (application interface block) 367
AIBTDLI interface to IMS 499
ALLOCATE command
  using with CALL command under TSO 74
  using with LOADGO command under TSO 76
AMODE
  assembler routines and 393
  C applications and 353
  C/C++AMODE considerations 12
  for CEEBXITA user exit 378
  heap storage 153
  in preinitialized routines 430
ampersand (&)
  how to specify in CEEXOPT 107
  using to pass by reference (indirect) 115
anywhere heap 150
apostrophe (’)
  using in CEEXOPT 107
application
  invoking MVS executable programs from a z/OS UNIX
  shell 81
  link-editing using c89 80
  placing MVS load modules in the z/OS UNIX file system
  81
  running
    from the z/OS UNIX shell 81
    under batch 82
    under MVS batch 82
application interface block (AIB) 367
AREA storage for PL/I 150
argc parameter for C
ARGPARSE runtime option
  possible combinations of runtime options and program
  arguments 105
ARGSTR 110
argument
  case sensitivity of arguments under TSO, for C
  applications 509
  distinguishing program arguments from runtime options
  104
  list format
    EXECOPS compiler option and 507
    EXECOPS runtime option and 505, 507
    how interactions of EXECOPS and PLIST compiler
    options affect 507
    how interactions of EXECOPS and PLIST runtime
    options affect 505
    PLIST compiler option and 507
    PLIST runtime option and 505
  passing
    by reference 114
    by value 114
    C passing for operating systems and subsystems
    115, 503
    directly 114
    guidelines for writing callable services 501
    indirectly 114
    relationship to parameter list 113
  specifying to an invoked routine which format to expect
    (C) 505
argv parameter for C
  C parameter passing styles and 505
arithmetic
  date calculations
    examples illustrating multiple calls to CEESECS
    callable service 291, 293
    overview 287
    examples using 345
asis 509
ASMTDLI interface to IMS 367
ASSEMBLE file 257
assembler language
  ASMTDLI interface 499
  C as substitute for 517
  COBOL parameter list format 509
assembler language (continued)
  EXEC DLI interface 499
macros
  CEECAA — generate a CAA mapping 404
  CEEEDSA — generate a DSA mapping 404
  CEEENTRY — generate a Language Environment-
    conforming prolog 400
  CEEFETCH — macro to dynamically load a Language
    Environment-conforming routine 409
  CEELOAD — macro to dynamically load a Language
    Environment routine 407
  CEEPPA — generate a PPA 404
  CEERELES — macro to dynamically release a
    Language Environment-conforming routine 416
  CEETERM — terminate a Language Environment-
    conforming routine 403
routines
  calling conventions for 393
  calling IMS PL/I routines from 368
  compatibility with Language Environment 391
  condition handling for 394
  equivalent callable services for 424
  example 421
  invoking callable services from 424
  main routines 393
  no support for assembler main routines under CICS
  392
  program check, handling 229
  subroutines 394
  system services available to 424
  system programming C considerations 520–522
  assigning
    message insert data 268
  assistive technologies 533
  atexit list
    CEEPIPI and 434
  ATTACH macro 426
  automatic data
    definition 139
    how used in enclave 141
B
below heap
  what used for 153
binder interface
  c89 utility 79
bit manipulation routines 343
BPXBATCH program
  invoking from TSO/E 82
  running an executable z/OS UNIX file under batch 82
building freestanding applications
  including alternate initialization routines for 517
  MVS 517
BYADDR compiler option 119
BYVALUE compiler option
  functions 119
  required if SYSTEM(CICS) specified 511
C
C
  AMODE/RMODE considerations 12
C (continued)
building linked list in 155
building system exit routines 520
calls to C under CICS 363
calls to C++ under CICS 363
case sensitivity of arguments under TSO 509
condition handling 183, 188
examples
CEE3CTY, CEEFMDT and CEEDATM 313, 314
CEE3CTY, CEEFMDT, CEE3MDS, CEE3MCS and
CEE3MTS 314
CEE3RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST
and CEEEDSP 159
CEEDAYS, CEEDATE and CEEODYWK 303
CEEFTST and CEEFRST 155
CEEHDLR, CEEGTST, CEECZST and CEEMCR 216, 217
CEEHDLR, CEEHLD, CEEQDT and CEEMCR 223
CEEMOUT, CEENCOD, CEEMGET, CEECOD and
CEEMSG 278
CEEQCN and CEEQSCN 288
CEESECS and CEEDATM 294
CEESECS, CEESEC, CEESEC, CEESEC, and CEEDATM 298
CEESECs, multiple calls to 291
coding main routine to receive inbound parameter
list 116, 119
freestanding MVS routines 518, 519
global condition handling model 183
interfaces to IMS from
list of DLI interfaces 499
LOADGO command considerations 76
LONGNAME compiler option 481
NOSTART compiler option 517
OPTIMIZE(2) compiler option 520
parameter passing, for operating systems and
subsystems
 PLIST and EXECOPS interactions 505, 507
styles 503
puts() function
freestanding routines and 518
RENT compiler option
making C routines reentrant with 122
specifying runtime options for
with _CEE_RUNOPTS 102
stderr
default destinations of 274
interleaving output with other output 275
redirecting output from 275
c89 utility
 interface to the linkage editor 79
 link-edit object modules 80
CALL command for TSO
ALLOCATE command and 74
case sensitivity of arguments when invoking C routine
with 509
example using 75
syntax description 74
CALL IMS interfaces 499
CALL statement
for COBOL
callable service feedback code and 235
callable services
CEE3ABD — terminate enclave with an abend 127
callable services (continued)
CEE3CIB — return pointer to condition information block
167
CEE3CTY — set default country 311
CEE3DMP — generate dump 337
CEE3GR — get the enclave return code 127
CEE3GRN — get name of routine that incurred condition
167
CEE3NG — set national language 311
CEE3MCS — get default currency symbol 311
CEE3MDT — get default decimal separator 311
CEE3MTS — get default thousands separator 311
CEE3PRM and CEE3PR2— query parameter string 127
CEE3SPM — query and modify Language Environment
hardware condition enablement 167
CEE3SRC — set the enclave return code 127
CEE3USR — set or query user area fields 337
CEECBLYD — convert date to COBOL Lilian format 285
CEECMI — store and load message insert data 257
CEECRHP — create new additional heap 147
CEECZST — reallocate (change size of) storage 147
CEEDATE — convert Lilian date to character format 285
CEEDATM — convert seconds to character timestamp
285
CEEDAYS — convert date to Lilian format 285
CEEDCOD — decompose a condition token 233
CEEDYOIK — calculate day of week from Lilian date 285
CEEFDMD — get default time format 311
CEEFMOD — get default time format and time format 311
CEEFLM — format monetary string 319
CEEFRST — free heap storage 147
CEEFOUT — format date and time into character string
319
CEEGMT — get current Greenwich mean time 285
CEEGLMTO — get offset from Greenwich mean to
time 285
CEEGPID — retrieve Language Environment version and
platform ID 337
CEEQDT — retrieve q_data_token 223
CEEGTJS 338
CEEFTST — get heap storage 147
CEEHDLR — register user condition handler 167
CEEISEC — convert integers to seconds 285
CEEJSON — move resume cursor relative to handle
cursor 167
CEELOCT — get current local time 285
CEELOCM — move resume cursor relative to handle
cursor 167
CEELOCM — move resume cursor relative to handle
cursor 167
CEESCLL — compare string collation weight 319
CEESEC — convert integers to seconds 285
CEESECS — convert timestamp to number of seconds
285
CEESCLL — signal a condition 167
CEESCLL — signal a condition 167
callable services (continued)

CEESTXF — transform string into collation weights 319
CEETEST — invoke debug tool 337
getting started with 339, 342
guidelines for writing 501
invoking under assembler 424
calloc() function 150

calls

dynamic call
C++ calling C++ under CICS 363
calls between COBOL and VS COBOL II, under CICS 363
external references resolved at run time when made 13

static call
external references resolved at run time when made 13
in CICS COBOL applications 363
casting, when using R1 and osplist macros 504
cataloged procedure
AFHWL (link-edit a Fortran program) 90
AFHWLG (link-edit and run a Fortran program) 90
AFHWLN (change external names in conflict between C and Fortran) 90
AFHWRL (separate and link-edit Fortran object module) 91, 122
AFHWRLG (separate, link-edit and run object module) 91, 122
AFHXFSTA (Fortran reentrancy program) 122
CBCC (compile a C++ program) 89
CBCCCL (compile, prelink, and link-edit a C++ program) 89
CBCCCLG (compile, prelink, link-edit, and run a C++ program) 89
CBCL (run a C++ program) 89
CBCLG (prelink and link-edit a C++ program) 89
CBCLP (relink and run a C++ program) 89
CEEWG (load and run an HLL program) 91
CEEWL (link-edit an HLL program) 92
CEEWLG (link-edit and run an HLL program) 92, 93
CXXFILT (invoke the demangle mangled names utility) 90

data set names 88
EDCC (compile a C program) 89
EDCCL (compile and link-edit a C program)
comparison with other cataloged procedures 88
using with system programming facilities 518
EDCCLG (compile, link-edit, and run a C program) 89
EDCCLIB (compile and maintain an object library) 90
EDCCLPLG (compile, prelink, link-edit, and run a C program) 88
EDDCSECT (invoke the DSECT conversion utility) 89
EDDLCDEF (invoke locale object utility) 89
EDDLCIB (invoke object library utility) 89
EDDCLP (prelink and link-edit a C program) 89
IBM-supplied
EDDCLIB 513
IEL1C (compile a PL/I program) 91
IEL1CG (compile, load, and run a PL/I program) 91
IEL1CL (compile and link-edit a PL/I program) 91
IEL1CLG (compile, link-edit, and run a PL/I program) 91
IGYWCG (compile, load, and run a COBOL program) 88
IGYWCL (compile and link-edit a COBOL program) 88

cataloged procedure (continued)
IGYWCLG (compile, link-edit, and run a COBOL program) 88
IGYWPL (prelink and link-edit a COBOL program) 91
introduction to 87
invoking 87
JCL and 60, 66
modifying 97
overriding and adding DD statements
nullifying parameters of 98
rules for 97
overriding and adding to EXEC statements 97
overriding default options 87
quick reference of 88
specifying runtime options in 69
step names in 87
unit names 88
CBCC cataloged procedure 89
CBCL cataloged procedure 89
CBCLG cataloged procedure 89
CBCLP cataloged procedure 89
CBCLPG cataloged procedure 89
CBLOPTS runtime option
VS COBOL II compatibility and 109
CBLPSHPOP runtime option
EXEC CICS PUSH and EXEC CICS POP commands and 361
CBLTDLI interface to IMS
list of DLI interfaces 499
CC CLIST
C370LIB parameter 515
CEE Facility_ID 270
CEE_RUNOPTS environment variable
specifying runtime options at invocation 102
CEE3ABD — terminate enclave with an abend
ABEND command and 424
CEE3CIB — return pointer to condition information block 168
CEE3CTY — set default country
examples using
CEE3CTY with CEEFMDT and CEEDATM 313, 316
CEE3CTY with CEEFMDT, CEE3MDS, CEE3MCS and CEE3MTS 314
CEE3DMP — generate dump
CEE transient data queue and 363
description 337
SNAP command and 426
CEE3GRN — get name of routine that incurred condition
examples using 226, 228
CEE3LNG — set national language
messages and 271
CEE3MCS — get default currency symbol
examples using 314
CEE3MDS — get default decimal separator
examples using 314
CEE3MTS — get default thousands separator
examples using 314
CEE3PRM and CEE3PR2— query parameter string 127
CEE3SPM — query and modify Language Environment
hardware condition enablement
advisory note regarding 501
condition handling, XUFLOW runtime option and 172
examples using 226, 228
CEE3USR — set or query user area fields 337
**CEEAUE_A_AB_CODES**

- specifying abend codes in 378

**CEEAUE_A_CC_PLIST** 382

**CEEAUE_A_OPTIONS** 382

**CEEAUE_A_WORK** 382

**CEEAUE_ABND** 382

**CEEAUE_ABTERM** 381

**CEEAUE_DUMP** 382

**CEEAUE_FBCODE** 383

**CEEAUE_FLAGS**

- CEEAU....ABND field of 382
- CEEAU....ABTERM field of 381
- CEEAU....DUMP field of 382
- CEEAU....STEPS field of 382

**CEEAUE_RETC**

- description 381
- relationship to CEEAU....ABND 381, 382

**CEEAUE_RSNCE**

- description 381
- relationship to CEEAU....ABND 382
- relationship to CEEAU....RETC 382

**CEEAUE_STEPS** 382

**CEEBINT** HLL user exit

- description 127
- functions 374
- interactions with CEEPIPI 434
- interface to 388
- languages it can be coded in 387
- terminating enclave created by 388
- user word parameter of, and CEEAU....USERWD 388

**CEEBLDTX** utility

- error messages 263
- using to create message files 257

**CEEBXITA** assembler user exit

- abends and
- requesting 377
- specifying codes to be percolated 378
- actions taken if errors occur within the exit 378
- AMODE/RMODE considerations 378
- application-specific 373
- behavior of
  - during enclave initialization 374, 375
  - during enclave termination 377
  - during process termination 378
- description 127
- EXEC CICS commands that cannot be used with 361
- functions 373
- IMS and 369
- installation-wide 373
- interactions with CEEPIPI 434
- interface to
  - diagram of 378
- modifications to, rules for making 378
- PLIRETC and 361
- specifying runtime options in 382
- TRAP runtime option and 378
- when invoked 375
- work area for 382

**CEECOD** — decompose a condition token

- examples using 278
- testing equivalent tokens 235

**CEEDOIT**

- specifying runtime options for MVS 69

**CEEDSA** assembler macro

- relationship to CEEENTRY 400
- syntax description 404

**CEEDSAZ** label 404

**CEEDYWK** — calculate day of week from Lilian date

- examples using 303

**CEEDY** assembler macro

- relationship to CEEENTRY 400
- relationship to CEEDSA 401
- relationship to CEEPPA 400, 401
- relationship to CEEETERM 400
- syntax description 400

**CEEFETCH** assembler macro 409

**CEEFMTD** — get default date format 311

**CEEFMDT** — get default date and time format

- examples using
- examples with CEE3CTY and CEE3MTS 314

**CEEGMT** — get current Greenwich mean time 286

**CEEGMTO** — get offset from Greenwich mean time to local time 286

**CEEGPID** — retrieve the Language Environment version and platform ID 337, 338

**CEEGQDT** — retrieve q_data_token

- examples using 223

**CEEHDLR** — register user condition handler

- assembler routines and 394
- condition handling example 188
- condition handling model and 178
- condition handling terminology 184
- examples using
  - assembler example 229, 232
  - CEE3SPM, CEE3GRN and CEEMOUT 226, 228
  - CEEGTST, CEECZST and CEEMRCR 216, 218
CEEHDLR — register user condition handler (continued)
examples using (continued)
examples with CEESGL, CEEGQDT and CEEMRCR 223
restrictions on using with various EXEC CICS commands 360, 424
SETRP command and 424
STAX command and 424
syntax description of user-written condition handlers 204, 206

CEEHDLU — unregister user condition handler
EXEC CICS HANDLE ABEND command and 424
SETRP command and 424
STAX command and 424
syntax description of user-written condition handlers 204, 206

CEEISEC — convert integers to seconds
examples using 298

CEELCNV — query locale numeric conventions
examples using 324, 326

CEELOAD assembler macro 407

CEELOCT — get current local time
examples using 287

CEELRR — initialize or terminate library routine retention 397

CEEMGET — get a message
examples using 278
relationship to condition tokens and other message services 234

CEEMOUT — dispatch a message
examples using 278
relationship to condition tokens and other message services 234
WTO command and 426

CEEMRCR — move resume cursor relative to handle cursor
examples using
examples with CEEHDLR, CEE3SPM and CEE3GRN 225, 228
examples with CEENCOD, CEEMGET, CEEDCOD and CEEMSG 278
relationship to condition tokens and other message services 234
resume action and 179

CEEMSG — get, format, and dispatch a message
examples using 278
relationship to condition tokens and other message services 234

CEENCOD — construct a condition token
examples using 278

CEEENTRY 400

CEEOPTS DD statement
restrictions 108
using 108

CEEPPA assembler macro
relationship to CEEENTRY 400
syntax description 404

CEEQCEN — query the century window
examples using 288, 291

CEEQDTC — query locale, date, and time conventions
examples using 327, 328

CEEQRYL — query active locale environment
examples using 331, 334

CEERELES assembler macro 416

CEEROPT description 101
specifying runtime options for MVS batch 69
CEESCEN — set the century window
examples using 288, 291

CEESCOL — compare string collation weight
examples using 329, 330

CEESECI — convert seconds to integers
examples using 298

CEESECS — convert timestamp to number of seconds
examples using CEESecs
C 291
COBOL 292
PL/I 293
examples with CEEDATM
C 294
COBOL 295
PL/I 297
examples with CEESCEN, CEESEC and CEEDATM
C 298
COBOL 299
PL/I 301

CEESETL — set locale operating environment
examples using 324, 328, 331, 332

CEESGL — signal a condition
ABEND command and 424
description of signals 170
examples using 223
EXEC CICS HANDLE ABEND command and 424
HLL-specific condition handlers and 171
relationship to condition tokens and message services 234
SETRP command and 424
STAX command and 424
TRAP runtime option does not affect 171
user-written condition handlers and 171

CEESTART preinitialization and 429
CEESTXF — transform string into collation weights
examples using 333, 334

CEETDLI interface to IMS
list of DLI interfaces 499

CEETERM assembler macro
relationship to CEEENTRY 400
syntax description 403

CEETEST — invoke debug tool
condition handling and 177

CEEUOPT
CEEUOPT ASSEMBLE 102
CEEEXOPT macro and 106
description 102
specifying runtime options for MVS 69

CEEWG cataloged procedure (load and run a program) 91

CEEWL cataloged procedure (link-edit a program) 92

CEEWLG cataloged procedure (link-edit and run a program) 92, 93

CEEXOPT macro
description 106
usage notes for 107

CEEXPIT macro 431

CEEXPITS macro 432

CEEXPITY macro 431

CESE transient data queue
CESE transient data queue (continued)

CEEMOUT and CEE3DMP output directed here 357
format 362
message handling and 273
CHAP command 424

CICS

callable service behavior under availability of callable services 357
CBLPSHPOP runtime option and 361
CESE transient data queue and 357
CICS region 351
CICS run unit
  behavior in nested enclave 470
  compared to Language Environment enclave 351
COBOL parameter list formats 509
coding main routines to receive parameters 118
condition handling for 359, 362
I/O restrictions in 352
link-editing for 353
list of interfaces to IMS that work from 499
message and dump output file 357
message format 362
message handling for 362
multitasking for 352
OS/VS COBOL compatibility considerations 357
PLIRETC support 353, 361
PLIRETV support 353
PLIST and EXECOPS interactions 505, 507
processing program table (PPT) 352
program control table (PCT) 352
reentrancy and 121
relinking PL/I applications 22
required level of 351
run-time output file 362
specifying runtime options for 354
storage and 358
SYSTEM setting 511
terminology 351
transaction 351, 352
transaction rollback 362
translator 352, 361

CLISTs for TSO
case sensitivity of args when invoking a C routine with 509
CMOD 73, 74
CLOSE command 426
CMOD CLIST 73, 74

COBOL

building a linked list in 155
can choose between static and dynamic calls under 13
condition handling 191, 194
constructing and dispatching a message for the significance condition 226, 228
examples
  CEE3CTY, CEEFMĐT and CEEDATM 314
  CEE3RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST
  and CEEDSHIP 159, 161
  CEEDAYS, CEEDATE and CEEDYWK 303, 305
  CEEFMON — format monetary string 320
  CEEFTDS — format date and time into character string 322
  CEEGTST and CEEFRST 155
  CEEDLR, CEE3SPM, CEE3GRN and CEEMOUT 225, 228

COBOL (continued)
examples (continued)
  CEEHDLR, CEEGTTST, CEECZST and CEEMRCR 218
  CEELCNV and CEESETL 324
  CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and
  CEEMSG 280
  CEEQCEN and CEESCEN 289, 290
  CEEQDTC and CEESETL 327
  CEECOL — compare string collation weight 329
  CEESCR and CEEDATM 295
  CEESCS, CEESEC1, CEESEC and CEEDATM 299
  CEESCS, multiple calls to 292
  CEESTL and CEEQRVL 331
  CEESTXF and CEEQRVL 333
coding main program to receive inbound parameters 116, 119

GOBACK statement
  generates return code 133
interfaces to IMS from
  list of DLI interfaces 499
ISPF 527
non-CICS OS/VS COBOL programs supported in single enclave only 470
order of program arguments and runtime options 109
OS/VS COBOL under CICS 357
parameter list formats 509
parameter passing style in Language Environment 115
preinitialization services 435
RENT compiler option 122
run-time options, specifying from 108
runtime options, specifying from 101

STOP RUN statement
  CEEBXITA assembler user exit and 375
effect SVC LINK has on 428
preinitialized environment and 434
return codes and 133
SVC LINK considerations for RES routines 428

comma (,) 104
coding a main routine to receive 116
command
  syntax diagrams xxv
command processor parameter list (CPPL)
  coding a main routine to receive 116
PLIST, EXECOPS and 505, 507
COMMON
  COBOL user-written condition handlers and 361
  common anchor area (CAA)
    writing assembler routines 393
  common environment, introduction 4
  compatibility
    assembler 391
    CICS 353
  compatibility, downward 9
  condition
    callable service feedback code and 233, 235
definition 170
divide-by-zero
    examples illustrating condition handling for 223
    nested 470, 473
severity
  CEEBXITA assembler user exit and 381
  COBOL condition handling 191
  condition token and 236
  ERRCOUNT runtime option and 174
  how to determine in a message 174, 270
condition (continued)
  severity (continued)
    TERMTHDACT runtime option and 177
    unhandled conditions and 135
condition handler
  C signal handlers
    CEEESGL callable service and 171
description 185
    TRAP runtime option and 171
description 178
  HLL semantics
    percolation and 179
    SORT and MERGE operations 524
    TRAP runtime option and 171
  PL/I ON-units
    CEEESGL callable service and 171
    SORT and MERGE operations 526
    TRAP runtime option and 171
user-written
  accessing a q_data structure and moving the
  resume cursor from 223
  C raise() function and 184, 185
  C signal() function and 185
  CEEESGL callable service and 171
coding 203, 206
  constructing message string when significance
  condition occurs 225
  EXEC CICS commands that cannot be used with
  360
  in ILC applications 207
  in nested condition handling 207
  introduction to user-written condition handlers 178
  registering with CEEHDLR callable service 178
  registering with USRHDLR runtime option 206
  role in Language Environment condition handling
  model of 178
  sort and merge operations and 524, 526
  syntax for 204
  TRAP runtime option and 171
condition handling
  assembler routines 394
  basic condition handling scenarios 179, 182
  C semantics
    default actions for C conditions 183
    example of 187, 188
    global error table and 183
    scenario of 186
    signal() function and 185
callable service feedback code and 233, 235
callable services for
  examples using CEEHDLR, CEEGTST, CEECZST and
  CEEMRRC 216, 218
  examples using CEEHDLR, CEEESGL, CEEQDT and
  CEEMRRC 223
  usage scenario 215
cICS, under 360
COBOL
  ON SIZE ERROR clause 191
  semantics of 191, 194
coding 203, 206
default actions 191
termination 208, 232
Fortran 195
Fprtram 194
condition handling (continued)
global model provided by C 183
IMS, under 394
introduction to 167, 179
nested enclaves
  created by C system() 474
  created by EXEC CICS LINK or EXEC CICS XCTL 470
  created by SVC LINK 472
  with a PL/I fetchable main 475, 476
PL/I 195, 197
  sort and merge considerations 524
  stack frame-based model provided by Language
  Environment
details of 179
  overview 167, 179
terminology 169, 170
  user exits and 362
  user-written condition handler 178
  using symbolic feedback code in 236, 241
  when to use 167
condition manager
  C signal handler and 188
  percolating abends and 191
  stack frame collapse and 193
  symbolic feedback code and 237
  thread initialization and 128
condition step 172, 174
condition token
  callable service feedback code and 233, 235
  condition handling model and 167
  messages and 270
  constructed reentrancy 121
contact
  z/OS 533
  continuations 107
  control block
    CAA 393
  COPY file 257
critical error message (severity 4) 270
cross system product (CSP) 504
csplist macro 504
ctdli() interface to IMS 367
CXT control block
  CEEAUE_A_AB_CODES field of 378
  CEEAUE_A_CC_PLIST field of 382
  CEEAUE_A_OPTIONS field of 382
  CEEAUE_A_WORK field of 382
  CEEAUE_FBCODE field of 383
  CEEAUE_FLAGS field of
    CEEAUE_DUMP field of 382
    CEEAUE_STEPS field of 382
  format of the 381
  CEEAUE_FUNC field of 380
  CEEAUE_LEN field of 380
  CEEAUE_USERWD field of
    user word parameter of CEEBINT and 388
CXXBIND EXEC 494
CXXFILT cataloged procedure 90
CXXMOD EXEC 494

D

data types
  guidelines for, when writing callable services 501
database rollback
  assembler user exit and Db2 365
  assembler user exit and IMS 369
  how CICS handles a 362
date and time
  services, summary
    TIME command and 426
Db2
  AMODE/RMODE considerations 12
DD statement
  data sets and
    defining data sets for the linkage editor 60
    defining data sets for the loader 68
    overriding in cataloged procedures 97–99
    proper format in JCL 69
Debug Tool
  C condition handling example of 188
  CEEBXITA and 378
deeding
  ABPERC runtime option and 171
definition side-deck 48
DELETE command
  EXEC CICS command 424
  host service 426
DELETE service routine for preinitialization
  components of 461
  return/reason codes for 461
DEQ 424
dereferencing 503, 504
DETACH 424
DFHECI (EXEC CICS interface stub) 353
DFHELII (EXEC CICS interface stub) 353
DFHPL101 (replaced by DFHELII) 353
DFSORT (Data Facility Sort)
  condition handling for 524, 526
  native invocations of 523, 524
  SVC LINK and 524
  user exits associated with 524
DISPLAY statement
  default file for 275
DL/I call 499
DLL code 38
DLLs (dynamic link libraries)
  application 38
  applications 38
  binding a DLL 48
  binding a DLL application 49
  C or C++ example 40
  calling explicitly 39
  calling implicitly 39
  COBOL/C example 42
  complex
    creating 56
    creating
      #pragma export 46
      C 45
      description 45
      exporting functions 46
    entry point 54
    example 53
    freeing 45
    function 37
    load-on-call 39
    loading 43
DLLs (dynamic link libraries) (continued)
  managing the use of 43
  performance 55
  restrictions 54
  sharing among application executable files 45
  using 50
  variable 38
downward compatibility 9
DSA (dynamic save area)
  register 13 and 393
dump
  CEEBXITA assembler user exit and 377, 382
  for CICS 357, 363
  Language Environment
    SNAP command and 426
dynamic call
  C, under CICS 363
  C++, under CICS 363
  external references resolved at run time when made 13
  VS COBOL II, under CICS 363

E
EDC facility ID 270
EDCC cataloged procedure 89
EDCCL cataloged procedure
  comparison with other cataloged procedures 88
  using in system programming C 518
EDCCLG cataloged procedure (compile, link-edit and run a C
  program) 88
EDCCLIB cataloged procedure (compile C program and
  invoke object library utility) 90, 513
EDCCLPLG cataloged procedure (compile, prelink, link-edit
  and run a C program) 88
EDCDECT cataloged procedure 89
EDCDELDEF cataloged procedure (invoke locale object utility)
  89
EDCCLIB cataloged procedure (invoke object library utility)
  89, 513
EIB (exec interface block)
  calls within same HLL and 363
  user-written condition handlers, EXEC CICS commands
  and 361
  enablement
    condition handling step
      definition of exceptions 170
      discussion of 170, 172
      TRAP runtime option and 171
enclave
  definition 141
  HLLs and 141, 142
  main routines and 141
  management of Language Environment resources 142
  multiple 142
  nested
    created by C system() function 469, 474
    created by EXEC CICS LINK or EXEC CICS XCTL
    469, 470
    created by SVC LINK 469, 471
    DFSORT and SVC LINK 524
    enclave with a PL/I fetchable main routine 475, 476
    MSGFILE ddnames and 273
  relationship with C main functions 141
  relationship with COBOL run units 141

549
examples (continued)

CEECLNV — query locale numeric conventions (continued)
with CEESETL 324, 326
CEEEMGET — get a message
with CEEEMOUT, CEEENCOD, CEEDCOD, and CEEMSG 226, 228
CEEEMOUT — dispatch a message
with CEEHDLR, CEE3SPM, and CEE3GRN 226, 228
with CEEENCOD, CEEMGET, CEEDCOD, and CEEMSG 225, 226
CEEEMSG — get, format, and dispatch a message
with CEEEMOUT, CEEENCOD, CEEMGET, and CEEDCOD 226, 228
CEENCED — construct a condition token
with CEEEMOUT, CEEMGET, CEEENCOD, and CEEMSG 226, 228
CCEEQCN — query century window 288
CEEQDTC — query locale, date, and time conventions
with CEESETL 327, 328
CEEQRYL — query active locale environment
with CEESETL 331, 332
with CEESTXF 333, 334
CEEESCEN — set century window 288
CEEESCL — compare string collation weight 329, 330
CEESECI — convert seconds to integers
with CEESECS, CEEISEC, and CEEDATM 298
CEESECS — convert timestamp to number of seconds
multiple calls to 291, 293
using CEEDATM with 294, 297
with CEESECI, CEEISEC, and CEEDATM 298
CEESETL — set locale operating environment
with CEECLNV 324, 326
with CEEQDTC 327, 328
CEEESIMOD — perform modular arithmetic 346
CEEESLOG — calculate logarithm base e 347
CEEESTXF — transform string into collation weights
with CEEQRYL 333, 334
ENTRY statement for MVS 517
INCLUDE statement for MVS
building freestanding MVS routine 518
including alternate initialization routines
MVS 517
invoking the prelinker
from TSO 492
LINK command 72
link-editing a fetchable load module 22
linking and running under TSO 71
LOADGO command 75
math services 345
overriding parameters in CEEWLG cataloged procedure 98
querying and setting the century window 288
relinking PL/I applications 22
EXCEPRTN service routine for preinitialization
components of 462
return/reason codes for 463, 464
exceptions
historical definition 170
Language Environment definition 170
EXEC CICS command
ABEND 360, 361, 382
DELETE 424
FREEMAIN 358, 426
GETMAIN 358, 426
EXEC CICS command (continued)
HANDLE ABEND
  assembler user exit and 361
  CEEHDLR callable service and 424
  CEEHDLU callable service and 424
  CEEESGL callable service and 424
table of equivalent Language Environment services 424
  TRAP runtime option and 360
  user-written condition handlers and 360
HANDLE AID
  assembler user exit and 361
  user-written condition handlers and 360
HANDLE CONDITION
  assembler user exit and 361
  user-written condition handlers and 360
IGNORE CONDITION 360, 361
LINK
  assembler routines and 426
  behavior of nested enclaves created by 470
  C and 363
  C++ and 363
  OS/VS COBOL and 364
  program management model and 352
  run-time options and 470
LOAD 424
POP HANDLE
  assembler user exit and 361
  user-written condition handlers and 360
PUSH HANDLE
  assembler user exit and 361
  user-written condition handlers and 360
RETURN 364
XCTL
  assembler routines and 426
  behavior of nested enclaves created by 470
  C and 363
  C++ and 363
  OS/VS COBOL and 364
  program management model and 352
  run-time options and 470
EXEC statement for MVS
  EXECOPS runtime option and 59
  invoking linkage editor 60
  invoking loader 66
  overriding, in cataloged procedures 97
  syntax for executing an application 68
  syntax for specifying runtime options 59
EXECOPS compiler option
  MVS argument list format and 507
EXECOPS runtime option
  CEENTRY macro and 402
  EXEC statements in JCL and 59
  MVS argument list format and 505
  TSO LOADGO command and 76
EXECs, IBM-supplied
  CXXMOD 494
executable files
  invoking MVS executable programs from a z/OS UNIX shell 81
  placing MVS load modules in the z/OS UNIX file system 81
  running from the z/OS UNIX shell 81
executable files (continued)
  running (continued)
    under batch 82
    under MVS batch 82
EXHIBIT for OS/VS COBOL
  default output file of 275
  no support for, under CICS 275
EXIT PROGRAM statement 426, 428
exit() function
  C condition handling scenario and 186
  CEEBINT HLL user exit and 388
  in a preinitialized environment 434
exporting functions 38
external data
  constructed reentrancy and 121, 122
  preinitialization and 432
  scope of, in Language Environment program management model 141, 142
Extra Performance Linkage (XPLINK) 25

F
facility ID
  each language component has a 269
  part of condition token 236, 269
  part of messages 270
feedback xxix
feedback code
  condition manager and 234
  condition token and 233, 235
  guidelines for writing callable services and 501
  in callable services 233
  omitting 235
  symbolic feedback code in condition handling 236, 241
fetch
  C fetching C 363
FETCH statement
  fetchable main
discussion of 475, 476
  reentrancy considerations of 476
  link-editing fetchable load modules 22
file
  executable
  placing MVS load modules in the z/OS UNIX file system 81
  running 80
fix-up and resume action
  compared to percolate, promote and resume actions 179
fork()
  MSGFILE runtime option and 273
Fortran
  AFHWL — link a Fortran program 95
  AFHWLG — link and run a Fortran program 96
  AFHWN — resolve Fortran and C name conflicts 96
  AFHWRLK — Fortran library replacement tool 14
  ARGSTR 110
  condition handling 195
  error message unit and Language Environment 276
  I/O statements, using 276
  library module replacement tool 14
  making Fortran programs reentrant 122
  order of program arguments and runtime options 110
  parameter passing style in Language Environment 110
Fortran (continued)
  replacing Fortran modules 14
  resolving Fortran and C name conflicts 96
  run-time options, specifying 108
  runtime options, specifying 101
  setting user return codes 134
  vector instruction exceptions 194
fprintf function 274, 275
fprintf function, 275
Fprtram
  condition handling 194
FREEMAIN 426
freestanding application
  alternate initialization routines for 517
  building
  MVS 517, 520
FREESTORE service routine for preinitialization
  components of 462
  return/reason codes for 462
freopen 275
function call for C 169
functions
  exported 38
  imported 38
G
genxlt
  CLIST 78
  utility
    TSO 78
GET command 426
GETMAIN 426
GETSTORE service routine for preinitialization
  components of 462
  return/reason codes for 462
global assembler user exit 373
global error table 183
H
HANDLE ABEND EXEC CICS command
  assembler user exit and 361
  CEEHDLR and 424
  CEEHDLU and 424
  CEESGL and 424
  table of equivalent Language Environment services 424
  TRAP runtime option and 360
  user-written condition handlers and 360
handle cursor
  promote action and 205
header files
  stdlib.h and the _R1 and _ospllist macros 504
  symbolic feedback code files and 237, 239
heap pool
  improving performance 151
heap pools
  applications which should use 152
  IBM-supplied defaults for CICS 356
  IBM-supplied defaults for non-CICS 105
  improve performance of heap storage allocation 151
  tuning heap storage 153
heap storage
  AMODE considerations of 153
  callable services for
    relationship to GETMAIN/FREEMAIN host services 424, 426
  examples of HLL data stored in 150
  heap element
    heap storage model and 150, 153, 162
  heap increment
    when allocated 150
  heap storage model 150, 154
  initial heap segment
    heap storage model and 153
    performance and 153
    when allocated 150
  leaks 151
  lifetime of 150
  program management model and 142
  RPTSTG runtime option and 153
  threads and 150
  tuning 153
  HEAPCHK runtime option
    use to identify storage leaks 151
  HEAPPOLS
    improving performance 151
I
IBM Facility_ID 270
iconv
  CLIST 78
  utility
    TSO 77
IEL1C cataloged procedure (compile a PL/I program) 91
IEL1CG cataloged procedure (compile, load and run a PL/I program) 91
IEL1CL cataloged procedure (compile and link-edit a PL/I program) 91
IEL1CLG cataloged procedure (compile, link-edit and run a PL/I program) 91
IGYWPL cataloged procedure (prelink and link-edit a COBOL program) 91
IGZ facility ID
  part of message 270
IGZERRE 429
ILBOSTP0 429
ILC (interlanguage communication)
  benefits of Language Environment support 167
  link-editing ILC applications 9
  overlay programs and 395
IMS (Information Management System)
  assembler calling PL/I under 368
  C considerations 367, 504
  CEETDLI interface 367
  coding a main routine to run under 118
  condition handling under 369
  link-edit considerations 368
  list of DLI interfaces 499
  OPTIONS(BYADDR) and 368
  OPTIONS(BYVALUE) and 368
  PLIST considerations
    PLIST and EXECOPS interactions 507
    PLIST(IMS) 118
  requirement for pre-loaded routines 121
  SYSTEM(IMS) compiler option and
IMS (Information Management System) (continued)
SYSTEM(IMS) compiler option and (continued)
how parameters are passed under 511
  table listing interfaces to 499
INCLUDE file 257
INCLUDE statement
  for MVS
    alternate initialization routines and 517
    application-specific assembler user exit and 374
    cannot use with loader 65
    freestanding applications and 518
    SYSLIB data set and 62
    SYSLIN data set and 60
    using linkage editor with 64
  informational messages 270
  initial heap
    heap storage model and 153
  initial stack segment
    performance and 149
initializing
  alternate initialization routines 517
  initialization routines 7, 62
  nested enclave
    CEEBXITA's function code for 381
    using CEEBXITA assembler user exit for
    CEEBXITA behavior 374
    function code for 380
input/output
  CICS restrictions 352
  Language Environment default message file attributes
    272, 273
insert data, message
  user-created
    assigning values to 268
installation-wide assembler user exit 373
interface validation exit 16
interleaved
  output 275
intrinsic functions 152
ISI (instance specific information)
  callable service feedback code and 236
    description 234
    q_data_token 244
ISPFA (Interactive System Productivity Facility)
  C/C++ AMODE/RMODE considerations 12
K
keyboard
  navigation 533
  PF keys 533
  shortcut keys 533
L
L-names
  LIBRARY control statement and 489
    mapping to S-names 490
  RENAME control statement and 489
  UPCASE prelink option and 496
Language Environment
  running
    applications under Language Environment 3
Language Environment (continued)
  summary of changes for V2R1 as updated February
    2015 xxx
  summary of changes for V2R2 xxx
  summary of changes for V2R3 xxx
library call processing
  data sets required by the linkage editor and 62
  LIBRARY statement and 64
  linkage editor options and 65
  NCAL link-edit option and 62, 64
library module replacement tool, Fortran 14
library routine retention 395
Library Search Order 69
LIBRARY statement
  cannot use with loader 65
  prelinker and 489
  using with linkage editor 64
Lilian date
  calculate day of week from (CEEDYWK) 285
  convert date to (CEEDAYS) 285
  convert to character format (CEEDATE) 285
  return current local date as a (CEELOCT) 285
  return GMT as a (CEEGMT) 285
LINK command for TSO
  example using 73
  how handled by Language Environment 426
  options for 77
  syntax description 72
link-editing
  application programs using c89 80
  for MVS
    CICS considerations 353
    diagram of linkage editor processing 59
    example of 62, 64
    IMS considerations 368
    input to the linkage editor 59
    options 64, 65
    using AFHWL cataloged procedure to link a Fortran
      program 95
    using AFHWLG cataloged procedure to link and run
      a Fortran program 96
    using CEEWL cataloged procedure to link a program
      92
    using CEEWLG cataloged procedure to link and run
      a program 92
    using INCLUDE statement to include additional
      modules as input 64
    using LIBRARY statement to specify additional
      libraries to be searched 64
    writing JCL for the linkage editor 60
  for TSO
    CMOD CLIST invokes LINK command 73, 74
    CXXMOD EXEC prelinks and links 494
    invoking the linkage editor using the LINK command
      72
    LINK-and-CALL method of processing 72
linkage editor
  cross-reference variables 64
  generate listing of control statements 64
  generating a module map 64
  input to 59
  invoking with the CMOD CLIST (TSO) 73, 74
  messages, where they go 64
linkage editor (continued)
module name 63
options for MVS
  LIST | NOLIST 64
  PRINT | NOPRINT 64
  RENT | NORENT 64, 122
  XREF | NOXREF 64
options for TSO
  CALL | NOCALL 77
  LET | NOLET 77
  PRINT(dat_set_name) 77
  RES | NORES 77
  SIZE(integer) 77
writing JCL for 60
LOAD service routine for preinitialization
components of 460
return/reason codes for 460
LOADGO command for TSO
  ALLOCATE command and 76
  EXECOPS runtime option and 76
  options for 77
  specifying runtime options in 76
  syntax description 75
loading
for MVS
diagram of loader processing 66
example of 68
input to the loader 65
options 67
standard data sets for 68
using CEEWG to load and run a program 91
writing JCL for the loader 66
for TSO
  LINK command syntax description 72, 73
  LOADGO command syntax description 76
  necessary data sets 76
  options 76, 77
library call processing
  data sets required by the loader and 68
local
data 142
locale callable services 319
LONGNAME compiler option 481
longname support 513
LPA (link pack area)
  reentrancy considerations 123
  RES loader option and 67, 77

M
macro
  _csplist 504
  _osplist 504
  _pcblist 504
  _R1 504
CEECAA
  relationship to CEEENTRY 400
  syntax description 404
CEEDSA
  relationship to CEEENTRY 400
  syntax description 404
CEEENTRY
  relationship to CEECAA 400

macro (continued)
CEEENTRY (continued)
  relationship to CEEDSA 401
  relationship to CEEPPA 400, 401
  relationship to CEETERM 400
  syntax description 400
CEEFETCH 409
CEELOAD 407
CEEPPA
  relationship to CEEENTRY 400
  syntax description 404
CEERELES 416
CEETERM
  relationship to CEEENTRY 400
  syntax description 403
CEEXOPT
  sample of CEEUOPT modified using 106
  usage notes for 107
CEEXPIT 431
CEEXPITS 432
CEEXPITY 431
main routine
  assembler main
    example of a simple 421
    example of main calling a sub 422
    register values on entry to 393
    determining 141
    nested enclave considerations 469
    position in Language Environment program
    management model 141
    preinitialization of 430, 437, 438
management of resources 142
management, program 139, 142
map heading 515
mapping
  L-names to S-names 490
math services
  about 343
  OS PL/I V2R3 23
member heading 516
MERGE (COBOL verb)
  condition handling considerations 524, 526
  overview 523
  user exit triggered by 524
message
  condition token and 269, 270
  directing to an I/O device 273
  example 270
  facility ID 269
  message prefixes 270
  severity
    codes and values 270
    using in your application 273
message file
  C stderr and stdout output and 275
  CICS considerations 273
  COBOL condition handling semantics and 191
  Fortran I/O statements 276
  Language Environment's default destinations 272
  nested enclave considerations 273, 479
  PL/I I/O statements 277
  specifying ddname of 273
  using CEEBLDCTX to assemble 257
message handling
message handling (continued)
  CESE transient data queue and 362
  relationship to fc parm of callable services 233, 235
  specifying dname of message file 273
message module table 257
  models, architectural
    program management 139, 142
MSGFILE runtime option
  COBOL condition handling semantics and 191
  default destinations under different operating systems 273
  different treatment under CICS 357
  POSIX runtime option and 273
  specifying ddnames across nested enclaves 273
  under z/OS UNIX 273
MSGRTN service routine for preinitialization
  components of 465
  return/reason codes for 466
multiple
  enclaves 142
  processes 141
MVS (Multiple Virtual System)
  running for
    specifying runtime options for 69, 102
    writing JCL to run an application 68, 69

N

NAB (next available byte)
  assembler main routine and 393
  assembler subroutine and 393
  CEEENTRY macro and 401
name conflicts, resolving
  between Fortran and C 15
  between static common blocks 15
  using AFHWN cataloged procedure 96
national language support (NLS)
  message handling and 271
natural reentrancy 121
natural reentrancy
  keyboard 533
nested conditions 207
nonoverrideable 383

O

object library utility
  adding object modules 513
  deleting object modules 513
  example under MVS batch 513
  listing the contents 513
  under batch 513
  under TSO 515
omitted parameter
  condition manager reaction to 235
  considerations when writing a callable service 501
ON EXCEPTION clause 191
ON SIZE ERROR clause 191, 194
OPEN command 426
OPTIONS(BYADDR) (continued)
  SYSTEM(CICS) and 511
  when it is the default 120
OPTIONS(BYVALUE)
  description 119
  IMS considerations 368, 511
  OPTIONS(NOEXECOPS) and 120
  rules for specifying 120
  specifying with OPTIONS(BYADDR) is an error 120
  SYSTEM(CICS) and 511
  when it is the default 120
OS ATTACH macro 426
ospist macro 504
overflow
  condition
    C SIGFPE condition and 183
    COBOL ON SIZE ERROR clause and 194
overlay
  programs 395
overrideable/nonoverrideable 383

P

parallel processing 142
parameter
  list
    accessing by using macros 503
    assembler 395
    relationship to argument list 113
  list format
    effect of EXECOPS compiler option on 507
    effect of EXECOPS runtime option on 505, 507
    how interaction of EXECOPS and PLIST compiler options affects 507
    how interaction of EXECOPS and PLIST runtime options affects 505, 507
    PLIST runtime option and 505, 507
  list pointer 115
  nullifying in cataloged procedures 98
  passing
    by reference 114
    by value 114
    C passing styles 503
    directly 114
    indirectly 114
  passing styles permitted by Language Environment 501

PARM statement 65
pblist macro 504
PCT (Program Control Table) 358
percolate action
  C condition handling and 186
  COBOL condition handling and 191
  compared to promote and resume actions 179
  condition handling model and 173
  user-written condition handler syntax for 205
persistent C environment 520
PL/I
  ALLOCATE statement 150
  BYADDR
    functions 119
  BYVALUE
    functions 119
PL/I (continued)

BYVALUE (continued)

must be specified if SYSTEM(IMS) or SYSTEM(CICS)
specified 511

condition handling 195, 197

Enterprise PL/I for z/OS

z/OS UNIX support same as for C++ 79

examples

CEE3CTY and CEEFMDT 161
CEE3CTY, CEEFMDT, CEEDATM 316
CEEEDAY, CEEDATE, CEEDWYK 305

CEEFMON — format monetary string 321

CEEFDS — format date and time into character
string 323

CEEGTST and CEEFRST 158
CEELCNV and CEESETL 326
CEEMOUT, CEENCOD, CEEMGET, CEEDCOD,
CEEMSG 282

CEEQCN and CEESCN 290

CEEQDTC and CEESETL 328

CEESCOL — compare string collation weight 330

CEESECS and CEEDATM 297

CEESECS, CEESCI, CEISEC, CEEDATM 301

CEESECS, multiple calls to 293

CEESELT and CEEQRYL 332

CEESTXF and CEEQRYL 334

coding main routines to receive inbound parm list
116, 119

FREE statement 150

guidelines for mixing PL/I and Language Environment
storage services 150

interfaces to IMS from

list of DLI interfaces 499

link-editing fetchable load modules 22, 23

linked list, building 158

MSGFILE considerations 278

OS PL/I V2R3 math services, using 23

parameter passing style 115

PLIBASE has been replaced 23

PLITASK no longer supported 23

REFER option 150

run-time options, specifying from 108

running in a non-IPT environment 84

runtime options, specifying from 101

SIBMBASE has been replaced 23

SIBMATH library has been added 23

storage in PL/I AREA 150

sysprint 363

SYSTEM compiler option

interactions with NOEXECOPS 511

variables, where stored 150

PLIRETC subroutine

CICS support for 353, 361

PLIRETV intrinsic function

CICS support for 353

PLIST compiler option

argument list format and 507

PLIST runtime option

argument list format and 505, 507

C interface to IMS 367

MVS setting and compatibility 505

POSIX

assembler user exit and 374

asynchronous interrupts 199

POSIX (continued)

communication

with COBOL or PL/I 199

condition token

for C-defined signals 242

for POSIX-defined signals 243

default signal action 132

EDC messages 258, 271

facility ID 271

IMS and 368

mapping

Language Environment abends to POSIX signals 190

5/370 exceptions to POSIX signals 189

messages 5201 to 5209 259

MSGFILE runtime option and 273

nested enclaves 469

position in Language Environment environment 4

process termination 132

running applications under Language Environment 3

signal handling

enabling or disabling signals 200, 201

handling, in the condition step 201

ILC considerations 199

signals that bypass condition handling 202

termination step for POSIX signals 202

synchronous interrupts 199

TERMTHDACT application 127

TERMTHDACT behavior under z/OS UNIX 202

using Language Environment services 3

POSIX runtime option 154

POST command 426

PPA (Program Prolog Area) 400, 404

PPT (Processing Program Table) 352

pragma

#pragma runopts

affecting argument list format with 505, 507

CICS and 354

IMS and 367

syntax description 103

preinitialization facility

benefits of 429

CEEPIPI(add_entry)

function code for 430

return codes from 453

syntax description 452

CEEPIPI(call_main)

assembler user exits and 444

CEEPIPI(init_main) and 444, 445

COBOL STOP RUN and 444

function code for 430

return codes from 445

CEEPIPI(delete_main_entry)

COBOL STOP RUN and 446

function code for 430

return codes from 447

syntax description 448

CEEPIPI(call_sub)

CEEPIPI(delete_main_entry)
preinitialization facility (continued)
CEEPIPI(delete_main_entry) (continued)
function code for 435
  return codes from 454
  syntax description 454
CEEPIPI(end_seq)
function code for 430
  return codes from 450
  syntax description 450
CEEPIPI(identify_entry)
function code for 435
  return codes from 455–457
  syntax description 455–457
CEEPIPI(init_main)
  CEEPIPI(call_main) and 444, 445
  CEEPIPI(term) 452
function code for 430
  return codes from 438, 439
  specifying service routines in 438, 439
  syntax description 437, 439
CEEPIPI(init_sub_dp)
function code for 430
  return codes from 443
  syntax description 443
CEEPIPI(init_sub)
  CEEPIPI(call_sub) and 444, 446
  CEEPIPI(term) and 452
function code for 430
  return codes from 441
  specifying service routines in 441
  syntax description 441
CEEPIPI(start_seq)
function code for 430
  return codes from 451
  syntax description 451
CEEPIPI(term)
  CEEPIPI(call_main) and 452
  CEEPIPI(init_main) and 452
function code for 430
  return codes from 452
  syntax description 451
CEESTART 429, 430
CEEXPIT macro 431
CEEXPITS macro 432
CEEXPITY macro 431
IGZERRE (COBOL interface to preinitialization) 429
ILBOSTP0 (COBOL interface to preinitialization) 429
old C interface to preinitialization and PLIST(MVS) 505
PIPI table
  add entry to 452
  CEEPIPI(call_main) and 445
  CEEPIPI(call_sub_addr) and 448
  CEEPIPI(call_sub) and 446, 448
  CEEPIPI(end_seq) and 450
  CEEPIPI(init_main) and 437, 439
  CEEPIPI(init_sub_dp) and 442
  CEEPIPI(init_sub) and 441
  CEEPIPI(start_seq) and 451
  generate entry within 431
  generate heading for 431
  identify end of 432
  introduction to 430
  restrictions against nested routines in 454
preinitialization facility (continued)
  service routines for
    allocating storage for 461
    AMODE/RMODE requirements of 460
    freeing storage of 462
    in CEEPIPI(init_main) 438, 439, 460
    in CEEPIPI(init_sub) 441, 460
    relationship to each other 460
    trapping program interruptions and abends 462
    vector format 459
prelinker
  CEEXMOD EXEC and 494
  constructed reentrancy 121
  freestanding MVS routine and 519
  functions 482
  how it maps L-names to S-names 490
  INCLUDE statement and 488
  invoking
    for TSO 492
    LIBRARY statement and 489
  prelink options 496
  prelinker map 485
  RENAME statement and 489
  when it has to be used 481
prelinking process 482
preventing in assembler 409
printf() function
  default destination 275
  interspersing messages into an application 274
process
  assembler user exit for termination of 381
  current support for 141
  definition 141
  relationship to enclaves 141
  role in Language Environment program management
    model 142
  termination of assembler routines and 394
Processing Program Table (PPT) 352
program
  link-editing using c89 80
  management model
    diagram of 142
    terminology of 141, 142
  placing MVS load modules in the z/OS UNIX file system
    81
  running under z/OS UNIX 80
program argument
  specifying with runtime options 104
program interrupts
  abend codes and return codes 136
  condition handling and 169, 171, 172
  q_data for 247
  SORT/MERGE and 524
  under CICS 360, 362
  under SORT/MERGE 524
  user exits and 378
Program Prolog Area (PPA) 400, 404
program specification block (PSB) 367, 368
prolog 400
promote action
  compared to percolate and resume actions 179
  condition handling model and 173
  user-written condition handler syntax for 205
PSB (Program Specification Block) 367
PUT command 426

Q
q_data
abends 244
arithmetic program interruptions 247
descriptor 254
math and bit-manipulation condition 250
square-root exception 250

R
R1 macro 504
raise() function for C
how C terminology differs from that of Language Environment 185
how C terminology differs from the Language Environment version 185
SIGTERM
HLL user exit and 388
random number service CEERANO 338
READ command 426
reason code
CEEPIPI(call_main) and 445
CEEPIPI(call_sub) and 447
in user exits 378, 381
summary of Language Environment codes 135
under CICS 361
recursion
allowed in user-written condition handlers 206
Language Environment program management model and 141
reentrancy
advantages of 121
C routines and
constructed reentrancy 121
natural reentrancy 121
procedure for generating reentrant load modules in 122
reentrant routines split into two parts 121
C Systems Programming Environment and 518
CEEPIPI(call_main) and 432
CICS routines and 121
COBOL RENT compiler option and 122
Fortran reentrancy separation tool 122
IMS and 369
making Fortran programs reentrant 122
modified CEEBXITA must be reentrant 378
MVS link pack area (LPA) and 123
PL/I REENTRANT compiler option and 123
pre-loaded IMS routines and 121
prelinker and 121
routines that must be reentrant 121
region (CICS) 351
RENAME control statement
how prelinkage utility maps L-names to S-names 490
syntax and usage notes 489
RENT compiler option
making C routines reentrant with 122
making COBOL programs reentrant with 122
prelinker must be used when C source file compiled with 481
resume
action
definition 178
severity 2 or above IGZ conditions and 191
user-written condition handlers and 191, 205
cursor
nested conditions and 207
return code
calculation 133
CEEAU_E_RET field of CEEBXITA and 381
CEEPIPI(call_main) and 445
CEEPIPI(call_sub) and 447
Fortran considerations 133
in user exits 381
possible C enclave return code incompatibility 133
RETURN-CODE special register 133
Return Code= nnn 266
Return Code= -1 263
Return Code= 0005 263
Return Code= 0006 263
Return Code= 0007 264
Return Code= 0008 264
Return Code= 0009 264
Return Code= 0010 264
Return Code= 0011 264
Return Code= 0020 264
Return Code= 0021 264
Return Code= 0028 264
Return Code= 0040 264
Return Code= 0044 264
Return Code= 0048 264
Return Code= 0052 265
Return Code= 0056 265
Return Code= 0060 265
Return Code= 0064 265
Return Code= 0068 265
Return Code= 0072 265
Return Code= 0076 265
Return Code= 0080 265
Return Code= 0084 265
Return Code= 0088 266
Return Code= 0092 266
Return Code= 0096 266
Return Code= 0098 266
Return Code= 0100 266
Return Code= 0104 266
Return Code= 0108 266
Return Code= 0112 266
return() 434
RMODE
C/C++ considerations 12
root
segment 395
RPTSTG runtime option
storage report generated by
using to tune the stacks 149
RTEREUS runtime option
preinitialization and 429
run unit
for CICS 351
for COBOL
relationship to Language Environment enclave 141
runtime environment, introduction 4
runtime options

558
runtime options (continued)
how nested enclaves get
enclaves created by C system() 474
enclaves created by EXEC CICS commands 470
enclaves created by SVC LINK 471
in the CEEPIPI interface to preinitialization 441, 445
in the user exit 377, 382
specifying
order of precedence 103
with program arguments 104

S
S-names
prelinker and
how L-names are mapped to S-names 490
saved segments
CEEPIPI and 437, 439, 441
SCEELKED link library
automatic call library and 62
cataloged procedures and
CEEWG 91
CEEWL 92
CEEWLG 92
changing library prefix of 98
MVS linkage editor procedures and 59, 62
MVS load procedures and 66, 68
TSO load/run procedures and 76
SCEERUN load library
CEEWG cataloged procedure and 91
CEEWLG cataloged procedure and 92
MVS load procedures and 66
TSO run procedures and 74
SCEESAMP sample library 362
search order
library for MVS 69
Search Order 69
sending to IBM
reader comments xxix
service routines
allocating storage for 461
AMODE/RMODE requirements of 460
freeing storage of 462
in CEEPIPI(init_main) 438, 439, 460
in CEEPIPI(init_sub) 441, 460
relationship to each other 460
trapping program interruptions and abends 462
vector format 459
SETRP command
CEEHDLR callable service and 424
CEEHDLU callable service and 424
CEESGL and 424
table of equivalent Language Environment services 424
severity of a condition
CEEBXITA assembler user exit and 381
COBOL condition handling 191
condition token and 236
ERRCOUNT runtime option and 174
how to determine in a message 174, 270
TERMTMHDACT runtime option and 177
unhandled conditions and 135, 174
short-on-storage condition 358
shortcut keys 533
SIGABRT
HLL user exit and 388
SIGTERM
HLL user exit and 388
slash (/)
specifying in PARM parameter 67
SNAP
CEE3DMP callable service and 426
table of equivalent Language Environment services 426
SORT/MERGE
condition handling within 524, 526
overview of sort/merge operations 523–525
user exits triggered by 524
stack
frame
condition management model and 172
differentiated from Global Error Table model of condition handling 183
getting 169
HLL-specific condition handlers and 178
stack frame zero 173, 177, 191
user-written condition handlers and 178
storage
Language Environment program management
model and 142
Language Environment stack storage model 148
RPTSTG runtime option and 149
tuning 149
stack and heap storage 147
STACK runtime option
using with RPTSTG to tune the stack 149
standard streams 275
static data 395
STAX command
CEEHDLR and 424
CEEHDLU and 424
CEESGL and 424
table of equivalent Language Environment services 424
stderr
default destinations of 274
STIMER command 426
STOP RUN
effect SVC LINK has on 426, 428
in a preinitialized environment 434
relationship to CEEPIPI(call_main) 444
STOP statement
for COBOL
CEEPIPI(call_sub) and 446, 447
in a preinitialized environment 434
storage
management model
heap storage 150, 162
heap storage leaks 151
stack storage 148, 150
manager 147
operating system services for 424, 426
service routines for 460
STORAGE built-in functions 359
storage tuning user exit 102, 103
subroutine
assembler
tables examples using 422
subroutine (continued)
   assembler (continued)
      register values of 393
      position in Language Environment program management model 141
      preinitialization and 430
      restriction regarding nested enclaves 469
success, testing a condition token for 234
summary of changes
   Language Environment
      V2R1 as updated February 2015 xxx
      V2R2 xxx
      V2R3 xxx
Summary of changes xxxi
SVC LINK 426, 428, 469, 471
switching 409
symbol information 516
symbolic feedback code 236, 242
syntax diagrams
   how to read xxv
SYSLIB
   linkage-editor and 59, 61, 62
   loader and 66, 68
SYSLIN
   linkage-editor and 59, 60
   loader and 65, 66, 68
SYSLMOD 59–61
SYSLOUT
   linkage-editor options and 64
   loader and 67, 68
SYSPRINT
   default destinations of MSGFILE runtime option 272
SYSPRINT
   linkage editor and 60, 61
   loader and 67, 68
SYSRCS 134
SYSRCT 134
SYSRCX 134
system programming facility, C
   benefits of 517
   building freestanding applications 517, 520
   persistent C environments 520
   reentrant modules 518
   summary of functions 520
   system exit routines 520
   user-server environments 520
SYSUT1 60, 61

T

TCB
   driven
      initialization of first enclave 374
      termination of first enclave 374
      nested enclaves and 374
termination
   causes under Language Environment 130
   CEETERM macro and 403
   enclave
      as indicated in CEEAUE_ABND field of CEEAUE_FLAGS 382
      as indicated in CEEAUE_ABTERM field of CEEAUE_FLAGS 381
   CEEBXITA behavior during 377
   (continued)
enclave (continued)
   CEEBXITA function codes for 381
   terminating enclave created by an assembler routine 394
   terminating enclave created by CEEBINT HLL user exit 388
   preinitialized routines and 429
   process
      CEEBXITA behavior during 378
      CEEBXITA function code for 381
      terminating process created by assembler routine 394
   TERMTHDACT runtime option and 177
   thread 176
termination imminent step
   discussion of 175, 177
TERMTHDACT runtime option
   condition message and 174
   POSIX runtime option and 202
   termination imminent step and 177
TEST runtime option
   condition handling model and 177
   thread
      IMS and 368
      role in Language Environment program management model 142
TIME command
   Language Environment date/time services and 426
   table of equivalent Language Environment services 426
trademarks 540
translator (CICS) 352, 361
TRAP runtime option
   ABPERC runtime option and 172
   CEEBXITA assembler user exit and abends that occur in CEEBXITA and 378
   using with CEEAUE_A_AB_CODES to percolate a list of abend codes 378
   CICS condition handling and 360
   errors occurring in CEEBXITA and 378
   how CEEAUE_ABND is affected by 382
   nested enclaves and
      enclaves created by C system() 474
      enclaves created by EXEC CICS LINK or EXEC CICS XCTL 470
      enclaves with a C or assembler main, created by SVC LINK 472
      enclaves with a COBOL main, created by SVC LINK 473
      enclaves with a PL/I fetchable main 475, 476
TSO (Time Sharing Option)
   running for
      ALLOCATE command and 74, 76
      CALL command and 74, 75
      LOADGO command and 75, 77
      specifying runtime options for 75–77, 102

U

user
   exit
      assembler 378
      for initialization 377, 434
      for termination 377, 378, 434
user (continued)
exit (continued)
  HLL 387
system exits in C Systems Programming
Environment 520
  under CICS 381, 382
under SORT/MERGE 524
heap (initial heap)
  heap storage model and 153
return code
  C language constructs that generate 133
  COBOL language constructs that generate 133
  Fortran language constructs that generate 134
  PL/I language constructs that generate 134
user comments 516
user interface
  ISPF 533
  TSO/E 533
user-server environment 520
user-written condition handler
  allowing nested conditions in 207
  as opposed to condition manager 178
  C raise() function and 184, 185
  C signal() function and
  terminology differences between C and Language
  Environment 185
CEESGL callable service and 171
coding 203, 206
examples 208, 232
EXEC CICS commands that cannot be used with 360
  in ILC applications 207
  in nested condition handling 207
  in SORT/MERGE condition handling 524
  registering with CEEHDLR 178
role in Language Environment condition management
model 178
syntax for 204
TRAP runtime option and 171
USRHDLR runtime option and 206
USRHDLR runtime option
description 206

V

variables
  exported 38
vector instruction exceptions 194

W

WAIT command 426
warning error message (severity 1) 270
working storage 142
writable static
  handled by prelinker 482
  writable static map 485
WRITE command 426
WTO command
  CEEMOUT callable service and 426
table of equivalent Language Environment services 426

X

XCTL command 426
XITPTR 378
XPLINK (Extra Performance Linkage)
  CEEXR cataluged procedure 93
definition 25
downward-growing stack 26
  glue code 27
  guard page 27
  how to enable 33
libraries
  SCEEBIND 8
  non-XPLINK application 26
parameter passing 31
register conventions 30
runtime option 33
stack overflow 30
when it should be used 32
XPLINK application 26
XPLINK environment 26
XPLINK stack 27
XPLINK stack frame layout 28
XPLINK (XPLINK)
  libraries
    SCEELIB 8
    when it should not be used 32
XUFLOW runtime option
using to manipulate the PSW 172

Z

z/OS Debugger
  CEEBINT and 388
condition handling model and 177
z/OS UNIX
  24-bit AMODE programs, restriction on using 81
c89 utility
    -c option 80
    -o option 80
    using 80
evironments supported 79
environments, application program
  shells through MVS batch 80
  TSO/E 80
link-editing
  C applications 79
  Fortran applications 79
linking and running, basic 79
MVS application program load module
  placing in file system 81
  shells through MVS batch 81
parent and child processes 132
PL/I MTF application support 86
PL/I support in a non-IPT environment 84
position in Language Environment environment 4
POSIX default signal action 132
POSIX process termination mapping 132
prelinking under 79
process termination 132
running
  C application program for MVS batch 81
  C applications 80
  Fortran applications 79
z/OS UNIX (continued)
running (continued)
  from the z/OS UNIX shell 81
  PL/I applications in a non-IPT environment 84
  PL/IMTF applications 86
  using BPXBATCH program 81
services 79
storage considerations 154
TERMTHDACT application 127