Language Environment
User's Guide

version 6 release 2
Contents

Figures .................................................... ix
Tables ....................................................... xi
About this document ................................. xiii
Unsupported z/OS functions ...................... xiii
Intended audience ................................ xiii
Where to find more information ................. xiv

How to send your comments to IBM .......... xv
If you have a technical problem .......... xv

Summary of Changes ................................. xvii
SA24-6199-01, z/VM Version 6 Release 2 .... xvii
  Language Environment upgrade for z/VM 620 x xvii
SA24-6199-00, z/VM Version 6 Release 1 .... xvii
SA24-6101-02, z/VM Version 5 Release 4 .... xvii
  Language Environment upgrade for z/VM 540 x xvii

Part 1. Language Environment Programming Guide .... 1

Chapter 1. Preparing to load and run under Language Environment .... 3
Understanding the Basics ........................... 3
Planning to load and run ............................ 3
Checking Which Run-Time Options Are in Effect .... 4
PL/I Considerations ................................ 4
  Replacing PL/I Library Routines in an OS PL/I Executable Program .... 4

Chapter 2. Loading and Running under z/VM .... 5
Basic Linking and Running ........................... 5
  Accepting the Default Run-Time Options .... 5
  Overriding the Default Run-Time Options ...... 6
Using the GLOBAL Command ....................... 6
  Resolving External References to Resident Routines .... 7
  Resolving External References to Dynamic Routines .... 7
C/C++ Considerations ................................ 8
COBOL Considerations ................................ 8
PL/I Considerations ................................ 8
  Determining the Search Order for Dynamic Routines .... 8
Using the LOAD and INCLUDE Commands ......... 9
  C/C++ Considerations ................................ 9
  PL/I Considerations ................................ 9
Using the LOAD Command ......................... 9
  Using the INCLUDE Command ..................... 10
Using the GENMOD Command ...................... 12
Using the BIND Command ......................... 13
Using the NUCXLOAD Command .................. 13
  Restrictions ....................................... 14
  Example ........................................... 14
Using FILEDEF to Define Input and Output Files ... 14
Link-Editing with the LKED Command ............ 14
Using the CMOD EXEC .............................. 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPLINK Considerations</td>
<td>61</td>
</tr>
<tr>
<td>COBOL Considerations</td>
<td>62</td>
</tr>
<tr>
<td>Determining the Behavior of Child Enclaves</td>
<td>62</td>
</tr>
<tr>
<td>Creating Child Enclaves by Calling a Second Main Program</td>
<td>62</td>
</tr>
<tr>
<td>Creating Child Enclaves Using SVC LINK or CMSCALL</td>
<td>62</td>
</tr>
<tr>
<td>Creating Child Enclaves Using the C system() Function</td>
<td>65</td>
</tr>
<tr>
<td>Creating Child Enclaves Containing a PL/I Fetchable Main</td>
<td>66</td>
</tr>
<tr>
<td>Other Nested Enclave Considerations</td>
<td>68</td>
</tr>
<tr>
<td>What the Enclave Returns from CEE3PRM</td>
<td>68</td>
</tr>
<tr>
<td>Finding the Return and Reason Code from the Enclave</td>
<td>70</td>
</tr>
<tr>
<td>Assembler User Exit</td>
<td>70</td>
</tr>
<tr>
<td>Message File</td>
<td>70</td>
</tr>
<tr>
<td>OpenExtensions Considerations</td>
<td>70</td>
</tr>
<tr>
<td>AMODE Considerations</td>
<td>70</td>
</tr>
<tr>
<td>Part 2. Language Environment Debugging Guide</td>
<td>71</td>
</tr>
<tr>
<td>Chapter 9. Debugging C/C++ Routines</td>
<td>73</td>
</tr>
<tr>
<td>Debugging C/C++ Input/Output Programs</td>
<td>73</td>
</tr>
<tr>
<td>__last_op Values</td>
<td>73</td>
</tr>
<tr>
<td>Using __errno2() to Diagnose Application Problems</td>
<td>76</td>
</tr>
<tr>
<td>Generating a Language Environment Dump of a C/C++ Routine</td>
<td>77</td>
</tr>
<tr>
<td>cdump()</td>
<td>77</td>
</tr>
<tr>
<td>csnap()</td>
<td>78</td>
</tr>
<tr>
<td>Chapter 10. Diagnosing Problems with Language Environment</td>
<td>79</td>
</tr>
<tr>
<td>Diagnosis Checklist</td>
<td>79</td>
</tr>
<tr>
<td>Part 3. Language Environment Run-Time Messages</td>
<td>81</td>
</tr>
<tr>
<td>Chapter 11. C/C++ Run-Time Messages</td>
<td>83</td>
</tr>
<tr>
<td>Chapter 12. COBOL Run-Time Messages</td>
<td>85</td>
</tr>
<tr>
<td>Part 4. Customizing</td>
<td>87</td>
</tr>
<tr>
<td>Chapter 13. Customizing</td>
<td>89</td>
</tr>
<tr>
<td>Updating Run-Time Options</td>
<td>89</td>
</tr>
<tr>
<td>Updating User Exit Options</td>
<td>90</td>
</tr>
<tr>
<td>C Component Locale Time Information</td>
<td>90</td>
</tr>
<tr>
<td>Updating Saved Segments</td>
<td>90</td>
</tr>
<tr>
<td>Updating the COBOL Component Reusable Environment</td>
<td>91</td>
</tr>
<tr>
<td>Modifying the behavior of the COBOL Reusable Environment</td>
<td>92</td>
</tr>
<tr>
<td>Part 5. Appendixes</td>
<td>93</td>
</tr>
<tr>
<td>Appendix A. Prelinking an Application</td>
<td>95</td>
</tr>
<tr>
<td>Which Programs Need to Be Prelinked</td>
<td>95</td>
</tr>
<tr>
<td>What the Prelinker Does</td>
<td>96</td>
</tr>
<tr>
<td>Prelinking Process</td>
<td>96</td>
</tr>
<tr>
<td>Primary Input</td>
<td>96</td>
</tr>
<tr>
<td>INCLUDE Control Statements</td>
<td>96</td>
</tr>
<tr>
<td>References to Currently Undefined Symbols (External References)</td>
<td>97</td>
</tr>
<tr>
<td>Processing the Prelinker Automatic Library Call</td>
<td>97</td>
</tr>
<tr>
<td>Language Environment Prelinker Map</td>
<td>98</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Control Statement Processing</td>
<td>102</td>
</tr>
<tr>
<td>IMPORT Control Statement</td>
<td>102</td>
</tr>
<tr>
<td>INCLUDE Control Statement</td>
<td>102</td>
</tr>
<tr>
<td>LIBRARY Control Statement</td>
<td>103</td>
</tr>
<tr>
<td>RENAME Control Statement</td>
<td>103</td>
</tr>
<tr>
<td>Mapping L-Names to S-Names</td>
<td>104</td>
</tr>
<tr>
<td>Starting the Prelinker</td>
<td>105</td>
</tr>
<tr>
<td>Examples</td>
<td>106</td>
</tr>
<tr>
<td>Prelinker Options</td>
<td>107</td>
</tr>
<tr>
<td>Appendix B. Parameter List Formats</td>
<td>109</td>
</tr>
<tr>
<td>C and C++ Parameter Passing Considerations</td>
<td>109</td>
</tr>
<tr>
<td>C PLIST and EXECOPS Interactions</td>
<td>111</td>
</tr>
<tr>
<td>Parameter Passing Considerations with XPLINK C and C++</td>
<td>112</td>
</tr>
<tr>
<td>COBOL Parameter Passing Considerations</td>
<td>113</td>
</tr>
<tr>
<td>PL/I Main Procedure Parameter Passing Considerations</td>
<td>113</td>
</tr>
<tr>
<td>Appendix C. Object Library Utility</td>
<td>115</td>
</tr>
<tr>
<td>Creating an Object Library</td>
<td>115</td>
</tr>
<tr>
<td>The LINKLOAD EXEC</td>
<td>116</td>
</tr>
<tr>
<td>Object Library Utility Map</td>
<td>118</td>
</tr>
<tr>
<td>Appendix D. Using the Systems Programming Environment</td>
<td>121</td>
</tr>
<tr>
<td>Building Freestanding Applications</td>
<td>121</td>
</tr>
<tr>
<td>Building Freestanding Applications</td>
<td>121</td>
</tr>
<tr>
<td>Special Considerations for Reentrant Modules</td>
<td>122</td>
</tr>
<tr>
<td>Building System Exit Routines</td>
<td>123</td>
</tr>
<tr>
<td>Building Persistent C Environments</td>
<td>123</td>
</tr>
<tr>
<td>Building User-Server Environments</td>
<td>123</td>
</tr>
<tr>
<td>Summary</td>
<td>124</td>
</tr>
<tr>
<td>Notices</td>
<td>127</td>
</tr>
<tr>
<td>Programming Interface Information</td>
<td>129</td>
</tr>
<tr>
<td>Trademarks</td>
<td>129</td>
</tr>
<tr>
<td>Glossary</td>
<td>131</td>
</tr>
<tr>
<td>Bibliography</td>
<td>133</td>
</tr>
<tr>
<td>Where to Get z/VM Information</td>
<td>133</td>
</tr>
<tr>
<td>z/VM Base Library</td>
<td>133</td>
</tr>
<tr>
<td>Overview</td>
<td>133</td>
</tr>
<tr>
<td>Installation, Migration, and Service</td>
<td>133</td>
</tr>
<tr>
<td>Planning and Administration</td>
<td>133</td>
</tr>
<tr>
<td>Customization and Tuning</td>
<td>133</td>
</tr>
<tr>
<td>Operation and Use</td>
<td>133</td>
</tr>
<tr>
<td>Application Programming</td>
<td>133</td>
</tr>
<tr>
<td>Diagnosis</td>
<td>134</td>
</tr>
<tr>
<td>z/VM Facilities and Features</td>
<td>134</td>
</tr>
<tr>
<td>Data Facility Storage Management Subsystem for VM</td>
<td>134</td>
</tr>
<tr>
<td>Directory Maintenance Facility for z/VM</td>
<td>134</td>
</tr>
<tr>
<td>Open Systems Adapter/Support Facility</td>
<td>134</td>
</tr>
<tr>
<td>Performance Toolkit for VM</td>
<td>135</td>
</tr>
<tr>
<td>RACF Security Server for z/VM</td>
<td>135</td>
</tr>
<tr>
<td>Remote Spooling Communications Subsystem Networking for z/VM</td>
<td>135</td>
</tr>
<tr>
<td>Prerequisite Products</td>
<td>135</td>
</tr>
<tr>
<td>Device Support Facilities</td>
<td>135</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Environmental Record Editing and Printing Program</td>
<td>135</td>
</tr>
<tr>
<td>Additional publications</td>
<td>135</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>137</td>
</tr>
</tbody>
</table>
## Figures

1. Example of a Message Source File .......................................................... 30
2. Location of User Exits ............................................................................. 43
3. Interface for CEEBXITA Assembler User Exit ........................................... 45
4. CEEAUE_FLAGS Format ........................................................................... 46
5. Format of Service Routine Vector ............................................................ 49
6. Assembler Driver That Creates a Preinitialized Environment .................. 57
7. Example of a Routine Using __errno2() .................................................... 77
8. Example of a Routine Using _EDC_ADD_ERRNO2 .................................. 77
9. Sample Output of a Routine Using _EDC_ADD_ERRNO2 .......................... 77
10. Customization EXEC - Panel 1 ................................................................. 89
11. Prelinker Map ......................................................................................... 99
12. Some Alternate C/C++ Parameter Passing Styles .................................... 109
13. Accessing Parameters Using Macros __R1 and __osplist ........................ 110
14. Examples of Casting and Dereferencing .................................................. 111
15. Object Library Utility Map ..................................................................... 118
16. Specifying Alternate Initialization at Link-Edit ....................................... 121
17. Simple Freestanding z/VM Routine ......................................................... 122
18. Building a Freestanding z/VM Routine ................................................... 122
19. Simple Reentrant Freestanding z/VM Routine ........................................ 122
20. Building a Reentrant Freestanding VM Routine ...................................... 122
About this document

This edition of the z/VM® Language Environment® User's Guide is intended to provide z/VM Language Environment users with information unique to the z/VM platform. The document is a supplement to the V1R9 z/OS® Language Environment manuals and should be used in conjunction with them.

This document is organized as follows:
- Part 1 describes Language Environment programming information unique to the z/VM platform. For more information regarding Language Environment programming, see [z/OS: Language Environment Programming Guide](#).
- Part 2 describes Language Environment debugging information unique to the z/VM platform. For more information regarding Language Environment debugging, see [z/OS: Language Environment Debugging Guide](#).
- Part 3 describes Language Environment run-time information unique to the z/VM platform. For more information regarding Language Environment run-time messages, see [z/OS: Language Environment Run-Time Messages](#).
- Part 4 describes Language Environment customizing information unique to the z/VM platform.
- The various appendixes describe prelinking, using parameter list formats, using the C object library, and systems programming environments.

Unsupported z/OS functions

The following major functions in the corresponding level of Language Environment on z/OS are not supported on z/VM:
- AMODE 64
- ASCII functions
- The following run-time options or parameters are not supported:
  - CEEDUMP(60,SYSOUT=*,FREE=END,SPIN=UNALLOC) - option not supported
  - DYNDUMP(*USERID,NODYNAMIC,TDUMP) - option not supported
  - HEAPCHK(OFF,1,0,0,0,0) - last parameter (0) not supported
  - HEAPPOOLS(OFF,8,10,32,10,128,10,1024,10,2048,10,0,10,0,10,0,10,0,10,0,10,0,10,0,10,0,10,0,10,0,10,0,10,0,10) - last 12 parameters not supported

Other differences in functionality will be noted in the appropriate documentation.

Intended audience

To use this document you should be familiar with the Language Environment product and one or more of the supported Language Environment-conforming high-level languages listed above. 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 run-time 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.

Where to find more information

For more information about z/VM functions, see the documents listed in the "Bibliography" on page 133.

Links to Other Online Documents

The online version of this document contains links to other online documents. These links are to editions that were current when this document was published. However, due to the nature of some links, if a new edition of a linked document has been published since the publication of this document, the linked document might not be the latest edition. Also, a link from this document to another document works only when both documents are in the same directory.
How to send your comments to IBM

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

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

1. Send an email to mhvrdfs@us.ibm.com.
2. Go to IBM z/VM Reader's Comments (www.ibm.com/systems/z/os/zvm/zvmforms/webqs.html).
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRDFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   U.S.A.
4. Fax the comments to us as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:
- Your name and address
- Your email address
- Your telephone or fax number
- The publication title and order number:
  z/VM V6R2 Language Environment User’s Guide
  SC24-6199-01
- The topic name or page number related to your comment
- The text of your comment

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

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

If you have a technical problem

Do not use the feedback methods listed above. Instead, do one of the following:
- Contact your IBM service representative.
- Contact IBM technical support.
- See IBM: z/VM Service Resources (www.ibm.com/vm/service/).
- Go to IBM Support Portal (www.ibm.com/support/entry/portal/Overview/).
Summary of Changes

This document contains terminology, maintenance, and editorial changes. Technical changes are indicated by a vertical line to left of the change. Some product changes might be provided through service and might be available for some prior releases.

SA24-6199-01, z/VM Version 6 Release 2
This edition supports the general availability of z/VM Version 6 Release 2 (z/VM V6R2).

Language Environment upgrade for z/VM 620
The Language Environment upgrade for z/VM 620 adds support to the equivalent of z/OS Language Environment 1.12. The upgrade will facilitate delivery of a new level of the C/C++ compiler. See "Unsupported z/OS functions" on page xiii.

SA24-6199-00, z/VM Version 6 Release 1
This edition supports the general availability of z/VM Version 6 Release 1 (z/VM V6R1).

SA24-6101-02, z/VM Version 5 Release 4
This edition supports the general availability of z/VM Version 5 Release 4 (z/VM V5R4).

Language Environment upgrade for z/VM 540
The Language Environment upgrade for z/VM 540 adds support to the equivalent of z/OS Language Environment 1.9. The upgrade will facilitate delivery of new levels of the C, COBOL, and PL/I compilers. See "Unsupported z/OS functions" on page xiii and "Using the CEEBLDCTX Utility" on page 30 for more information. Additional information on customization has been added in Chapter 13, "Customizing," on page 89.
Chapter 1. Preparing to load and run under Language Environment

This chapter discusses z/VM specific information that you need to know before loading 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 load different libraries, the procedure is similar to that used in pre-Language Environment versions of C, COBOL, or PL/I. For more information about running applications under Language Environment, see \textit{z/OS: Language Environment Programming Guide} and the individual language migration guides.

\textbf{Restriction:} Language Environment does not support Fortran applications in the z/VM environment.

Understanding the Basics

Language Environment library routines are divided into two categories: \textit{resident routines} and \textit{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 reload 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 load process combines output from compilers, language translators, load programs and control statements to produce an executable program (load module or program object) and stores it in a library. The executable program can then be run from that library. Either the program management binder or linkage editor can be used to perform the load 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 \textit{z/VM: Program Management Binder for CMS} for a complete discussion of services to create, load, modify, list, read, transport and copy executable programs.

Planning to load and run

There are certain considerations for z/VM that you must be aware of before loading and running applications under Language Environment. They are:

- Language Environment resident routines, including those for callable services, initialization, and termination, are located in the following libraries:
  - SCEELKED TXTLIB - for non-XPLINK C application programs
  - SCEECPP TXTLIB - for non-XPLINK C++ application programs
  - SCEEBND2 TXTLIB - for XPLINK C and C++ application programs
- Language Environment dynamic routines are located in relocatable CMS MODULEs and SCEERUN LOADLIB. The relocatable CMS MODULEs can be installed as nucleus extensions or in shared segments.
Checking Which Run-Time Options Are in Effect

Using the Language Environment run-time option RPTOPTS, you can control whether a run-time options report is produced; with the Language Environment run-time option MSGFILE, you can control where report output is directed.

RPTOPTS generates a report of all the run-time 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:

- In a POSIX (ON) application it goes to file descriptor 2.
- In a POSIX(OFF) application if you override the default setting of RPTOPTS, a report is sent to the FILEDEF specified by SYSOUT unless you override the MSGFILE run-time option to specify a different location. The default destination for any MSGFILE output is the TERMINAL unless you change it by issuing a FILEDEF for the file specified in the MSGFILE option.

If you want to change the options report destination, you can alter the default setting of the MSGFILE run-time option, which specifies where all run-time diagnostics and messages are written. For example, if you specify MSGFILE(OPTRPRT) and RPTSTG(ON), the storage report is written to a file whose ddname is OPTRPRT. The default runtime options can also be customized. See “Updating Run-Time Options” on page 89 for more information.

For the syntax of RPTOPTS and MSGFILE, see z/OS: Language Environment Programming Reference.

PL/I Considerations

The information that follows is additional for use with z/VM when using z/OS: Language Environment Programming Guide.

Replacing PL/I Library Routines in an OS PL/I Executable Program

Under VM, you can use the PL/I library replacement tool IBMWRLK TEXT (a member of SCEELKED TXTLIB) to replace the OS PL/I library routines in your OS PL/I executable programs with the analogous Language Environment resident routines. The executable programs must be created with the LKED command and reside in CMS LOADLIBs. It is not possible to replace run-time library routines in a file of type MODULE created by the GENMOD command.

For further information on library routine replacement, see PL/I for MVS & VM Compiler and Run-Time Migration Guide.
Chapter 2. Loading and Running under z/VM

Before you can run a program under z/VM, you must issue one of the following commands:

- LOAD (Stores a copy of the program in virtual storage)
- GENMOD (Stores the program on disk)
- LKED (Stores the program in a LOADLIB)
- BIND (Stores the program on disk using the Program Management binder)

z/VM produces an object module with the file type TEXT when you compile your program. Before you run the program, external references inserted by the compiler must be resolved. Use one of the following methods to create an executable application; you can run your application after you complete any of these steps.

- Create a temporary copy of your program in virtual storage by using LOAD and INCLUDE commands. No permanent copy of the executable program is made.
- Create a module using LOAD, INCLUDE, GENMOD, and/or BIND commands if appropriate. A module is an executable application that is stored as a file with a file type of MODULE.
- Create a module in a member of a library using the LKED command. This method link-edits an executable application and stores it as a load module in a member of a CMS LOADLIB.
- Create a module using the CMOD EXEC (C applications only). See "Using the CMOD EXEC" on page 15 for more information.

OpenExtensions has its own section on linking, loading, and running C applications and PL/I routines in those applications (see Chapter 3, “Building, Loading, and Running under OpenExtensions,” on page 23).

Restrictions:

- VisualAge® PL/I and Enterprise PL/I are not supported on z/VM. Language Environment continues to provide support for PL/I applications under z/VM compiled with PL/I for MVS & VM, and previous, supported levels of the PL/I compiler.
- Language Environment does not support Fortran applications in the z/VM environment.
- Enterprise COBOL for z/OS and OS/390®
- COBOL programs compiled with the DLL or ARITH(EXTEND) compiler options are not supported on z/VM.
- COBOL programs using object oriented constructs, LINE SEQUENTIAL files, or dynamic allocation using environment variables are not supported on z/VM.
- COBOL multi-threaded or multi-tasking programs are not supported on z/VM.

Basic Linking and Running

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

Accepting the Default Run-Time Options

Use the following series of CMS GLOBAL, LOAD, and START commands to accept default run-time options:
GLOBAL TXTLIB SCEELKED
GLOBAL LOADLIB SCEERUN
LOAD MYPROG
START *

This series of commands does the following:

- Identifies text libraries that you want z/VM to search to resolve external references in your object code, including the Language Environment SCEELKED (text) link library, and any libraries where your text files are located
- Links one or more text files containing object code and loads them into storage
- Runs the image of the application that is assembled in storage by the LOAD command.

Overriding the Default Run-Time Options

If MYPROG is a C/C++ or a PL/I application that uses routines from SCEELKED and MYTXTLB and wants to send Language Environment MSGFILE output (including the options report) to file OPTRPRT, issue the following commands:

GLOBAL TXTLIB SCEELKED MYTXTLB
LOAD MYPROG
GLOBAL LOADLIB SCEERUN
FILEDEF OPTRPRT DISK OPTRPRT OUTPUT A
START * RPTOPTS(ON), MSGFILE(OPTRPRT)

If MYPROG above is a COBOL application, then you need to modify the START command to be:

START * / RPTOPTS(ON), MSGFILE(OPTRPRT)

For more information, see "Using the GLOBAL Command," "Using the LOAD and INCLUDE Commands" on page 9, and "Using the START Command" on page 18.

Using the GLOBAL Command

You must issue a GLOBAL command before using the CMS LOAD command and before running applications. The syntax of the GLOBAL command is:

Syntax

GLOBAL LOADLIB libname1...libname63
TXTLIB
other keywords

LOADLIB

Specifies the load module libraries to be searched for a module that the OSRUN command or the LINK, LOAD, ATTACH, or XCTL macros refer to. The libraries can be CMS LOADLIBs or OS module libraries. If you specify an OS data set, issue a FILEDEF command for the data set before you issue the GLOBAL command.

TXTLIB

Specifies the text libraries to be searched for missing subroutines when the LOAD or INCLUDE command is issued, when the LKED command is issued, or when a dynamic load occurs (that is, when an OS SVC 8 or SVC 122 is issued).

Subroutines that are dynamically loaded should contain only VCONs that are resolved within the same text library member or that are resident in storage.
throughout the processing of the original CMS LOAD or INCLUDE command. Otherwise, the entry point is unpredictable.

**other keywords**
Additional GLOBAL keywords which do not apply to loading or running an application under z/VM and are, therefore, not shown here.

**libname1 - libname63**
The file names of up to 63 libraries of the specified file type (LOADLIB or TXTLIB). The libraries are searched in the order in which they are named. The library list is subject to other system limits, such as command line length. This command supersedes any previous GLOBAL command for the specified file type. If no file names are specified, the command cancels any previous GLOBAL command for this file type.

### Resolving External References to Resident Routines
A GLOBAL TXTLIB command must be issued for the Language Environment text library to resolve external references to the Language Environment resident routines before a CMS LOAD command is issued. Before loading an application, issue the following command:

```
GLOBAL TXTLIB SCEELKED usertxt
```

- SCEEBND2
  - The Language Environment C++ text library for XPLINK application programs.

- SCEECPP
  - The Language Environment C++ text library for non-XPLINK application programs.

- SCEELKED
  - The Language Environment text library.

- usertxt
  - The name of any user-generated text library or libraries to be searched for text files needed by your application.

### Resolving External References to Dynamic Routines
Before running your application, you must issue a GLOBAL LOADLIB command; this enables the Language Environment LOADLIB to resolve external references to the Language Environment dynamic routines:

```
GLOBAL LOADLIB SCEERUN userload
```

- SCEERUN
  - Identifies the Language Environment load library.

- userload
  - The name of any user-generated load library or libraries to be searched for load modules needed by your application.

Check with your system administrator to find out where Language Environment dynamic routines are located at your installation. In addition to a set of relocatable load modules and the load library SCEERUN LOADLIB, some Language Environment routines might have been installed in a nucleus extension or a saved segment.
C/C++ Considerations

If your C/C++ application performs long double arithmetic or uses extended-precision arithmetic, you must also specify the CMSLIB text library in your GLOBAL TXTLIB command. You can combine the CMSLIB with other TXTLIBs, as follows:

GLOBAL TXTLIB SCEELKED CMSLIB usertext

In addition to specifying CMSLIB, the C/C++ application must be run with TRAP(ON,SPIE).

COBOL Considerations

To run OS/VS COBOL programs, you must specify the SCEERUN and SCEEILBO libraries on the GLOBAL LOADLIB command.

PL/I Considerations

The product structure for PL/I has changed from previous versions and most CMS EXECS that load a PL/I application using the OS PL/I library must be changed to include SCEELKED, SIBMATH, or SIBMCALL.

- SCEELKED contains the stubs for PL/I library routines, in addition to Language Environment-conforming languages and Language Environment-provided routines and stubs.
- SIBMATH contains the stubs for old OS PL/I V2R3 math library routines.
- SIBMCALL provides PLICALLA and PLICALLB compatibility for PL/I for MVS & VM applications that use OS PL/I PLICALLA or PLICALLB as an entry point.

SIBMCALL and SIBMATH libraries must be concatenated before SCEELKED. They can be concatenated in any order.

For example, if your PL/I application requires OS PL/I math support, you must specify the SIBMATH library. In link-edit steps, this library must precede SCEELKED if old math results are needed in a particular load module. You can combine the SIBMATH with other LOADLIBs as follows: GLOBAL TEXTLIB SIBMATH SCEELKED usertext

SIBMATH
The Language Environment load library, containing the stubs for old OS PL/I V2R3 math library routines.

SCEELKED
The Language Environment text library.

usertext
The name of any user-generated text library or libraries to be searched for subroutines needed by your application.

Determining the Search Order for Dynamic Routines

The search order for dynamically loaded routines is:
1. Nucleus extension
2. Saved segments
3. Relocatable load modules
4. Load modules in LOADLIBs
5. Object modules
6. TXTLIB members
Normal CMS search order prevails when searching for a particular type in the previous list. Files on the A-disk are searched before files on the B-disk.

In general, the sooner a dynamically loaded routine is found, the better the performance of an application. For overall system performance gains, it is better to place heavily used dynamically loaded routines into a saved segment where they can be shared by all users.

Using the LOAD and INCLUDE Commands

The loader is invoked by using the LOAD command that reads one or more TEXT files (containing relocatable object code) or members of a text library from a minidisk or directory and loads them into virtual storage. LOAD establishes proper linkages between the files. The file containing the main routine should be the first file named in the command unless you specify the entry point name on the RESET option. The syntax of the LOAD command is:

Syntax

```
LOAD filename1 filename2 ... filename_n [ (options) ]
```

filename
Name of a file you want to load into storage.

options
List of LOAD options separated by blanks or commas (see Table 1 on page 10 for a list of available options).

Specify the RLDSAVE option for the LOAD command if you intend to use GENMOD. For more information about LOAD and its options, see VM/ESA: CMS Command Reference.

C/C++ Considerations

If the main routine is C/C++, specify the following under the options for the LOAD command:

`RESET CEESTART`

PL/I Considerations

If the main procedure is PL/I for MVS & VM, specify `RESET CEESTART` under the options for the LOAD command. For more information about using the LOAD command with PL/I, see PL/I for MVS & VM Compiler and Run-Time Migration Guide.

Using the LOAD Command

The following example causes the text library containing Language Environment resident routines, SCEELKED, and the USERTXT text library to be searched for files that your application needs to run. The files PROGRAM1 and CEEUOPT are loaded into virtual storage and a load map is written as follows:

```
GLOBAL TXTLIB SCEELKED USERTXT
LOAD PROGRAM1 CEEUOPT (MAP
```
## VM/CMS Load Options

Table 1 contains a selection of CMS load options.

### Table 1. Selected CMS Load Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESET</strong></td>
<td>RESET sets the starting location for the applications currently loaded.</td>
</tr>
<tr>
<td>entry</td>
<td>The <em>entry</em> name must be an external name (for example, a CSECT control section or ENTRY) in the loaded applications.</td>
</tr>
<tr>
<td>*</td>
<td>If you specify *, the results are the same as if the RESET option were omitted. If the RESET option is omitted, the default entry point is used.</td>
</tr>
<tr>
<td><strong>MAP</strong></td>
<td>MAP writes a load map to a file in your minidisk or directory named LOAD MAP A5.</td>
</tr>
<tr>
<td>NOMAP</td>
<td>NOMAP specifies that no LOAD MAP file is created.</td>
</tr>
<tr>
<td><strong>TYPE</strong></td>
<td>TYPE displays the load map at your terminal and writes it to a file on minidisk or directory.</td>
</tr>
<tr>
<td>NOTYPE</td>
<td>NOTYPE does not display the file at your terminal.</td>
</tr>
<tr>
<td><strong>LIBE</strong></td>
<td>LIBE searches text libraries for missing subroutines. The text libraries must be previously defined by a GLOBAL command.</td>
</tr>
<tr>
<td>NOLIBE</td>
<td>NOLIBE does not search text libraries for unresolved differences.</td>
</tr>
<tr>
<td><strong>START</strong></td>
<td>Runs the application when loading has completed.</td>
</tr>
<tr>
<td><strong>NORLDSAV</strong></td>
<td>NORLDSAV instructs the CMS loader not to save relocation information from the TEXT files being loaded.</td>
</tr>
<tr>
<td>RLDsave</td>
<td>Specify RLDsave if you plan to use the GENMOD command. RLDsave instructs z/VM to save relocation information from the text files. The GENMOD command uses relocation information to generate relocatable CMS modules.</td>
</tr>
<tr>
<td><strong>AMODE</strong></td>
<td>Specifies the addressing mode of the application in a 370-XA mode virtual machine. In a System/370 mode virtual machine, you can specify AMODE, although only 24-bit addressing is available. This allows you to create XA capable module files on an S/370 mode virtual machine. Valid AMODE values are:</td>
</tr>
<tr>
<td>24</td>
<td>The entry point of the application receives control in 24-bit addressing mode.</td>
</tr>
<tr>
<td>31</td>
<td>The entry point of the application receives control in 31-bit addressing mode when running on a 370-XA mode virtual machine, and in 24-bit addressing mode on a System/370 mode virtual machine.</td>
</tr>
<tr>
<td>ANY</td>
<td>The entry point is capable of operating in either 24- or 31-bit addressing mode.</td>
</tr>
</tbody>
</table>
Table 1. Selected CMS Load Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMODE 24</td>
<td>ANY</td>
</tr>
<tr>
<td>24</td>
<td>The load module must reside below the 16M line in a 370-XA virtual mode machine.</td>
</tr>
<tr>
<td>ANY</td>
<td>The load module must reside above the 16M line in a 370-XA virtual mode machine.</td>
</tr>
</tbody>
</table>

| HOBSET | HOBSETSD | NOHOBSET | Specifies if the high-order bit for V-type constants (VCONs) of SD (CSECTs) or LD (ENTRYs) types is to be turned on or left unchanged. This option only applies to PL/I applications. |

Note: Set loader tables (LDRTBLS) to a minimum of 6.

**HOBSET, HOBSETSD, and NOHOBSET PL/I Options**

PL/I programs that execute with the CMS LOAD and INCLUDE commands can specify the HOBSET, HOBSETSD, or NOHOBSET option. The default is NOHOBSET.

**HOBSETSD and HOBSET:** PL/I programs that execute with the CMS LOAD and INCLUDE commands can contain entry addresses whose high-order bit is set if the referenced name has the AMODE 31/ANY attribute. This applies to:

- Both external CSECT and external label names, if the HOBSET option is in effect
- External CSECT names only, if the HOBSETSD option is in effect

The following considerations apply when using the HOBSETSD or HOBSET options:

- Entry variables and constants generated by compiled code can have entry addresses whose high-order bit is set. A PL/I program can access such addresses by using:
  - The ENTRYADDR or UNSPEC builtin/pseudovariable
  - The PL/I BASED or DEFINED language construct that allows entry variables to be overlaid
  - Assembler routines that receive or pass the addresses

Entry addresses that have the high-order bit set should be used with care, such as in the following situations:

- Because the high-order bit can be set in such addresses, comparison to the PL/I NULL() value should be avoided.
- It might be necessary to preserve the high-order bit of entry addresses to ensure that entry variables are built correctly using the ENTRYADDR pseudovariable.
- If an external procedure entry name is referenced within the external procedure itself, comparisons involving these entry references might produce unexpected results. This constraint applies only to the HOBSET option.

**NOHOBSET:** The high-order bit of entry addresses is always zero with this option. You can use this option without any of the restrictions and precautions described in "HOBSETSD and HOBSET."
Using the INCLUDE Command

The LOAD command loads a TEXT file or member of a text library into virtual storage. The INCLUDE command loads additional TEXT files or members of a text library that make up your executable application.

The INCLUDE and LOAD commands have similar formats and option lists. The main difference is that if you issue two LOAD commands in succession, the second command replaces the first. The INCLUDE command, on the other hand, cannot be used unless you have just issued a LOAD. You can specify as many INCLUDE commands as necessary following the LOAD command to load files into storage. The files specified in the INCLUDE command must refer to subroutines. See Table 1 on page 10 for a list of available options.

The syntax of the INCLUDE command is:

```
INCLUDE filename1 filename2 ... filename_n [options]
```

**filename**
Name of a file you want to include into storage.

**options**
List of INCLUDE options separated by blanks or commas (for a list of available options, see Table 1 on page 10).

For more information about INCLUDE and its options, see VM/ESA: CMS Command Reference.

The following example loads a TEXT file from the USERTXT text library and includes another TEXT file from another text library into the load module. A load map is also written.

```
GLOBAL TXTLIB SCEELKED USERTXT USERTXT2
LOAD PROGRAM1
INCLUDE PROGRAM2 (MAP
```

Using the GENMOD Command

Use the GENMOD command with the LOAD and INCLUDE commands to create application modules, that is, relocatable files whose external references have been resolved. In z/VM, these files must have a file type of MODULE. The syntax of the GENMOD command is:

```
GENMOD filename [options]
```

The GENMOD command takes a copy of the executable module in virtual storage and stores it onto a disk with a `filename` that you specify. In the following example, PROGRAM1, PROGRAM2, and PROGRAM3 are TEXT generated from C source files that are put into a module with a file name of PROGRAM1 and a file type of MODULE:
GLOBAL SCELLKED USERTXT
LOAD PROGRAM1 PROGRAM2 PROGRAM3 (RLDSAVE RESET CEESTART
GENMOD PROGRAM1 (NOMAP

If you use the name of an existing module, the previous version of the module is replaced. If you do not specify in filename the name of the file where you want the load module to be stored, the GENMOD command processor defaults to the first entry point in the load map.

If the main entry point module is a PL/I or C application, load the object modules into storage using the RLDSAVE option and issue the GENMOD COMMAND using the FROM CEESTART option.

For ILC applications that dynamically load or fetch other programs, you must specify the RLDSAVE option of the LOAD statement and the NOMAP option of the GENMOD statement.

After you create the module with GENMOD, run the application composed of the source files PROGRAM1, PROGRAM2, and PROGRAM3 by entering:

PROGRAM1

Using the BIND Command

Use the BIND command to create application modules, that is, relocatable files whose external references have been resolved. In VM, these files must have a file type of MODULE. The syntax of the BIND command is:

Syntax

BIND filename [ ( options [ ) ] ]

The BIND command takes a copy of the executable module in virtual storage and stores it onto a disk with a filename that you specify. In the following example, PROGRAM1, PROGRAM2, and PROGRAM3 are TEXT files that are put into a module with a file name of PROGRAM1 and a file type of MODULE:

GLOBAL TXTLIB SCELLKED USERTXT
BIND PROGRAM1 PROGRAM2 PROGRAM3

If you use the name of an existing module, the previous version of the module is replaced. If you do not specify in filename the name of the file where you want the load module to be stored, the BIND command processor defaults to the first entry point in the load map.

After you create the module with BIND, run the application composed of the source files PROGRAM1, PROGRAM2, and PROGRAM3 by entering:

PROGRAM1

Using the NUCXLOAD Command

Use NUCXLOAD to load the modules into storage and install them as nucleus extensions. You can use the NUCXLOAD command if the RLD information has been saved during the CMS LOAD command using the RLDSAVE option.
Running under z/VM

Restrictions

Only reentrant modules can be installed as nucleus extensions. In PL/I for MVS & VM, use the REENTRANT procedure option; in COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, or VS COBOL II, use the RENT compiler option to ensure that a module can be installed as a nucleus extension. OS/VS COBOL modules cannot be installed as nucleus extensions because OS/VS COBOL cannot generate reentrant code.

C/C++ object modules that contain L-names or that are not naturally reentrant must be compiled with the RENT option and prelinked before being installed as nucleus extensions.

Example

In the following example, a CMS MODULE is created from a PL/I program, PROGRAM1. The TEXT file is loaded into storage and the RLD information is saved during the CMS LOAD using the RLDSAVE option. For PL/I, the GENMOD command requires the FROM CEESTART option as shown below:

GLOBAL TXTLIB SCEELKED
LOAD PROGRAM1 (RLDSAVE RESET CEESTART
GENMOD PROGRAM1 (NOMAP FROM CEESTART
NUCXLOAD PROGRAM1
PROGRAM1

For more information about the LOAD, GENMOD, and NUCXLOAD commands, see z/VM: CMS Commands and Utilities Reference and VM/ESA: CMS User's Guide.

Using FILEDEF to Define Input and Output Files

If your program requires input and/or output files, you must define these files using the CMS FILEDEF command prior to executing the module. The FILEDEF command relates the ddname of the input or output file specified in your program with an I/O device. For example, if PROGRAM1 contains a ddname of an input file stored on your A disk as MYDATA INPUT, issue the following command (infile is the ddname of the input file specified in PROGRAM1):

FILEDEF infile DISK MYDATA INPUT A

For more information about the GENMOD and FILEDEF commands, refer to VM/ESA: CMS Command Reference.

Link-Editing with the LKED Command

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

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

After you issue the appropriate FILEDEF commands, issue the LKED command.
Syntax

\[\text{LKED} \text{-} \text{filename} \]

\[
\begin{align*}
\text{NAME} & \text{ membname} \\
\text{LIB} & \text{ libname}
\end{align*}
\]

\[\text{NAME} \text{ membname} \]

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

\[\text{LIB} \text{ libname} \]

Name of the LOADLIB file where the resulting load module is placed.

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

\text{FILEDEF SYSLIB DISK SCEELKED TXTLIB E}
\text{LKED PROGRAM1 (NAME PROGRAM1 LIB USERLOAD}

For more information about the LKED command and a complete list of options, see \textit{VM/ESA: CMS Command Reference}.

Using the CMOD EXEC

The IBM-supplied CMOD EXEC invokes the loader or the binder (depending on the parameters passed or the compiler being used), which loads one or more object modules into virtual storage, resolves external references, and creates an executable module with the file type of MODULE. This EXEC can be used by C/C++ applications only. The syntax of the CMOD EXEC is:

Syntax

\[\text{CMOD} \text{-} \text{textdeck} \]

\[
\begin{align*}
\text{option}
\end{align*}
\]

\[\text{textdeck} \]

Name of the input text decks; the file type must be TEXT.

\[\text{options} \]

Options you want to apply as the executable module is being generated. The options are listed in \textit{Table 2}.

\textbf{Table 2. CMOD options}

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binder specific options</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2. CMOD options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINDOPTS(options)</td>
<td>Specifies options for the Binder. These options may be any of the options supported by the Binder.</td>
</tr>
<tr>
<td>C++</td>
<td>Specifies that at least one of the text decks is C++. This must be specified for C++ code to be correctly linked.</td>
</tr>
<tr>
<td>DLL(side file name(s))</td>
<td>If a side file name is not specified, this just passes the DYNAM DLL option to the Binder. It is the same as specifying BINDOPTS(DYNAM DLL), which enables the module for dynamic linking. A definition side file will be produced with the same name as the first text deck name, and a file type of SYSEFDSD. If a side file name is specified, the DYNAM DLL option is still passed to the Binder, but also the Binder will process the definition side file specified. An 8 character CMS file name is specified. CMOD will look for that file name with a file type of SYSEFDSD. Multiple names can be specified, separated by blanks.</td>
</tr>
<tr>
<td>XPLINK</td>
<td>Specifies that the text deck(s) has been compiled with the XPLINK option. Generally speaking, XPLINK text decks cannot be bound with non-XPLINK text decks.</td>
</tr>
</tbody>
</table>

#### LOAD/GENMOD/Prelinker specific options

| AMODE                  | Specifies the addressing mode in which the program will be entered in a virtual machine. For a complete description of AMODE, refer to the LOAD command in the CMS command reference manual. |
| AUTO|NOAUTO                | Specifies that your disks are to be searched for TEXT files for use in resolving undefined references. |
| CPLINK(options)         | Specifies options for the Prelinker. |
| DUP|NODUP                 | Specifies that an error message is to be generated if duplicate CSECT names are encountered. If you want to ensure that only one copy of a object module is loaded, use the NODUP option. |
| GENMOD(options)         | Passes any options to the GENMOD command. |
| INV|NOINV                 | Specifies that invalid card images are not to be included in the load map. |
| LET|NOLET                 | Specifies that all LOAD errors for the load module are to be ignored and an attempt to generate a module will be made. |
| ORIGIN                 | Specifies where CMS loads the program. This location must be in the CMS transient area or in any free CMS storage. |
| RLD|INWORLD               | Specifies that relocation directory information is to be saved in the load module. |
| STR|INOSTR                | Specifies that storage is to be initialized during the generation of the executable module. |
| RMODE                  | Specifies where the program is to reside in a virtual machine with greater than 16MB of storage. For a complete description of RMODE, refer to the LOAD command in the CMS command reference manual. |

#### Common options

| MAP|NOMAP               | The specified option is passed to the Binder or the LOAD command. For MAP (which is the default), the Binder will incorporate a module map into the SYSPRINT output; the LOAD command will generate a load map file on your A disk with the name LOAD MAP A. |
Running under z/VM

Table 2. CMOD options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODNAME modulename</td>
<td>The default is to generate an executable module having the same file name as the first object module specified, a file type of MODULE, and a file mode of A. The MODNAME option allows you to give a specific name to the executable module. Specify the module name (modulename) immediately following the MODNAME keyword, and the CMOD EXEC creates an executable module named modulename MODULE A.</td>
</tr>
</tbody>
</table>

Using the LINKLOAD EXEC

Use the IBM-supplied LINKLOAD EXEC to produce the fetchable C/C++ members in a CMS load library. For more information, see [z/OS XL C/C++ Programming Guide](#). The LINKLOAD EXEC is used only by C/C++. The syntax of the LINKLOAD EXEC is:

**Syntax**

```
LINKLOAD textdeck (LIB libname option)
```

*textdeck*  
Name of input text decks. The file type of the text decks must be TEXT and the source code must contain a `#pragma linkage(name, FETCHABLE)` preprocessor directive.

Do not specify the file type or file mode when using this EXEC.

*option*  
Options you want to apply as the fetchable load module is being generated. The options are listed in Table 3

Table 3. LINKLOAD Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB libname</td>
<td>A keyword used to indicate that the next argument, libname, is the name of the library where the load member is to be stored. The library name parameter must be specified, but if it is the first parameter, the keyword LIB is optional.</td>
</tr>
<tr>
<td>CPLINK (options)</td>
<td>Allows you to pass options to the prelinker. CPLINK is called if it is required by the text decks or if a CPLINK option is given. See Appendix A, “Prelinking an Application,” on page 95 for more information.</td>
</tr>
<tr>
<td>MBR memname</td>
<td>A keyword specifying that the next argument, memname, is the name of the member within the load library that is to be generated. If you do not specify a memname, the name of the text deck containing the fetchable code is used.</td>
</tr>
</tbody>
</table>
Running under z/VM

Table 3. LINKLOAD Options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Specifies that the load member generated by LINKLOAD is to be added to the load library. If a member by the same name already exists, the new member is not added.</td>
</tr>
<tr>
<td>REPLACE</td>
<td>Specifies that the load member generated by LINKLOAD is to replace a member by the same name in the load library. If the member does not already exist, the new member is added.</td>
</tr>
<tr>
<td>NEW</td>
<td>Specifies that an existing load library of the same name containing only the named member is created. The existing load library is replaced by the new library containing only the named member.</td>
</tr>
</tbody>
</table>

LKED (options) Allows you to pass options to the LKED command. For more information on the LKED command, see "Link-Editing with the LKED Command" on page 14.

Using the START Command

After you load your application into virtual storage with the LOAD command and issue the appropriate GLOBAL commands, use the CMS START command to execute your application. The syntax of the START command is:

**Syntax**

```
START [program_parameter_string]
```

* Specifies that control passes to the application’s default entry point at execution time. This option is required if program parameters are passed. (For information about how the default entry point is determined, see z/OS: Language Environment Programming Guide) You can also indicate an entry point whose name you specify in the application.

**program_parameter_string**

Specifies the run-time options and program parameters passed to the main routine in the application. The run-time options and program parameters are normally separated by a slash but C/C++ and PL/I users must omit the slash (unless it is part of a program parameter) if the NOEXECOPS run-time option is in effect.

In the following example, the compiled program PROGRAM1 is loaded and run with the RPTSTG(ON) and RPTOPTS(ON) run-time options specified:

```
GLOBAL TXTLIB SCEELKED USERTXT
LOAD PROGRAM1
GLOBAL LOADLIB SCEERUN
```
Running under z/VM

START * RPTSTG(ON),RPTOPTS(ON)/

Note: For COBOL programs, the "/" must be specified first by default.

In the case of an application for which you do not supply any run-time options or parameters, you can load and execute by using the START option of the LOAD command: LOAD PROGRAM1 ( START

In C/C++, you can use EXECOPS in the #pragma runopts directive to enable the passing of run-time options in the START command. If NOEXECOPS is similarly specified, any run-time options specified on the command line are treated as program parameters. For more information on how to specify run-time options, see z/OS: Language Environment Programming Guide.

Using the iconv Utility and ICONV EXEC 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 output code page. There is one record in the output file for each record in the input file. No padding or truncation of records is performed.

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

The ICONV EXEC invokes the iconv utility, which copies the input file to the output file and converts the characters from the input code page to the output code page. It can be invoked under VM/CMS or z/VM batch.

For information on the iconv utility, see XL C/C++ for z/VM: User’s Guide.

Using the genxlt Utility and GENXLT EXEC for C/C++

The genxlt utility reads character conversion information from the input file and writes the compiled version to the output file. The input file contains directives that are acted upon by the genxlt utility to produce the compiled version of the conversion table.

The GENXLT EXEC invokes the genxlt utility, which reads the character conversion information and produces the conversion table. It can be invoked under VM/CMS or z/VM batch. For information on the genxlt utility, see XL C/C++ for z/VM: User’s Guide.

Running Your Application

You can run an application under z/VM after you have issued one of the following commands:

- LOAD (Stores a copy of the program in virtual storage)
- GENMOD (Stores the program on disk)
- LKED (Stores the program in a LOADLIB)
- BIND (Stores the program on disk using the Prgram Management binder)
Running under z/VM

Running a Module Produced by the BIND or GENMOD Command

After you create a module using the GENMOD command and have issued the GLOBAL LOADLIB SCEERUN command, you can execute the module. Enter the module name on the command line and, optionally, pass the module both run-time options and parameters, as shown in the syntax below.

Syntax

```
modname program_parameter_string
```

```
program_parameter_string
```

Specifies the run-time options and program parameters passed to the main routine in the application. The run-time options and parameters that are passed to the main routine are normally separated by a slash. Run-time options and program parameters are discussed in the z/OS: Language Environment Programming Guide.

In the following example, the GENMOD command is extended to show run-time options. In this example, PROGRAM1, PROGRAM2, and PROGRAM3 are TEXT files that are put into a module with a file name of PROGRAM1 and a file type of MODULE. PROGRAM1 is executed by typing its name on the CMS command line with a list of run-time options that you want to pass to it as follows:

```
GLOBAL TXTLIB SCEELKED USERTXT
LOAD PROGRAM1 PROGRAM2 PROGRAM3 (RLDSAVE RESET CEESTART
GENMOD PROGRAM1 (NOMAP
GLOBAL LOADLIB SCEERUN
PROGRAM1 RPTSTG(ON),RPTOPTS(ON)/
```

To use the GENMOD command, the LOAD is performed using the RLDSAVE option in this example. PL/I and C require the RESET CEESTART option of the LOAD command and the FROM CEESTART option of the GENMOD command. The slash at the end of the last line of this example is required for C/C++ or PL/I, except when NOEXECOPS is in effect. The slash needs to be at the beginning of the line for COBOL.

Running a Module Using the OSRUN Command

After you create a module and store it in a LOADLIB using the LKED command, you can run it using the OSRUN command. Before running your module, you must issue a GLOBAL command to identify to z/VM the LOADLIB containing the module, plus the Language Environment LOADLIB to identify Language Environment load modules that are called by your application. The syntax of the OSRUN command is:

Syntax

```
OSRUN memname PARM=program_parameter_string
```

```
memname
```

Member name containing the load module you created using LKED. The member name is in turn located in the load library that you identified to z/VM using the GLOBAL command.

```
program_parameter_string
```

Specifies the run-time options and program parameters passed to the main
routine in the application. The run-time options and program parameters are
normally separated by a slash but C/C++ and PL/I users must omit the slash
(unless it is part of a program parameter) if the NOEXECOPS run-time option is
in effect.

Program parameters should be enclosed in single quotation marks since
run-time parameters pass special characters.

For example, if you wanted to run a C or PL/I program named PROGRAM1, and
you wanted to specify the RPTSTG(ON) and RPTOPTS(ON) run-time options, you
would issue the following commands:

FILEDEF SYSLIB DISK SCEELKED TXTLIB E
LKED PROGRAM1 (NAME PROGRAM1 LIBE USERLOAD
GLOBAL LOADLIB SCEERUN USERLOAD
OSRUN PROGRAM1 PARM='RPTSTG(ON),RPTOPTS(ON)/'

Using the VM/CMS Extended Parameter List

When z/VM transfers control to an application, the CMS extended parameter list is
used to construct the main routine’s parameters and get run-time options. Language
Environment repackages the CMS extended parameter list according to the format
Environment:

• An application with a COBOL main program receives the CMS extended
  parameter list as a halfword-prefixed string.
• An application with a C/C++ main routine receives the CMS extended parameter
  list in an argc, argv format.
• If your PL/I application specifies SYSTEM(CMS), you receive the CMS extended
  parameter. If your application specifies SYSTEM(CMSTPL), however, it receives
  the CMS tokenized parameter list.
Running under z/VM
Chapter 3. Building, Loading, and Running under OpenExtensions

The interface to the CMS module build facilities for OpenExtensions C/C++ applications is the OpenExtensions c89 utility. You can use c89 to compile and build an OpenExtensions C/C++ program in one step, or bind application object modules after the compilation. You can run the c89 utility from either the OpenExtensions shell or directly from CMS. For more information on using the c89 utility, see z/VM: OpenExtensions Commands Reference.

Note: VisualAge PL/I routines are not supported under OpenExtensions. Therefore the PL/I information in this chapter applies only to earlier versions of PL/I.

PL/I for MVS & VM routines are supported under OpenExtensions. PL/I for MVS & VM routines can run in the non-IPT without any unique restrictions other than those described in PL/I for MVS & VM Compiler and Run-Time Migration Guide. PL/I routines can run in the non-initial thread (non-IPTs) created by C routines with some restrictions. Limited PL/I – C ILC is supported in non-IPTs.

Basic Building and Running C/C++ Applications under OpenExtensions

OpenExtensions supports the following environments for running your OpenExtensions C/C++ applications:
- OpenExtensions shell
- CMS

Using the OpenExtensions-supplied utility c89, you can compile and build an OpenExtensions C/C++ application in one step, or bind application object modules separately. To produce an executable file, invoke c89 and pass it object modules (file.o BFS files or CMS native files) without using the -c option. For information on the c89 utility, see z/VM: OpenExtensions Commands Reference.

Invoking the OpenExtensions Shell

To begin a shell session, you first log on to z/VM as a CMS user and then invoke the shell with the OPENVM SHELL command. The shell and all processes and process groups running under it are typically in the same session. For more information about starting a shell session, see z/VM: OpenExtensions Commands Reference.

Using the OpenExtensions c89 Utility to Create Executable Files

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

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

- To build an application program object file to create the default executable file `a.out` in the working directory, specify:
  ```sh
c89 usersource.o
```
- To build an application object file to create the `mymod.out` executable file in the `app/bin` directory, relative to your working directory, specify:
  ```sh
c89 -o app/bin/mymod.out usersource.o
```
  where `usersource.o` is the object file created by compilation with `c89`.
- To build several application object files to create the `mymod.out` executable file in the `app/bin` directory, relative to your working directory, specify:
  ```sh
c89 -o app/bin/mymod.out usersrc.o othersrc.o
```
- To build an application object file to create the `myloadmd` module file on the A disk specify:
  ```sh
c89 -o //myloadmd.module usersource.o
```
- To compile and build an application source file with several `zinfo` in the `approg/lib` subdirectory, relative to your working directory, specify:
  ```sh
c89 -o approg/lib/zinfo usersrc.c existobj.o //pgmobj.module
```

The `c89` utility specifies default values for some prelinker and module build options. It also passes prelinker and module build options by using the `-W` option. For more information on using the `c89` options, see [z/VM: OpenExtensions Commands Reference](#).

Prelinker Options

With the exception of the OE option described below, the other prelinker options are the same as described in Appendix A, “Prelinking an Application,” on page 95.

**OE|NOOE**

The OE option causes the prelinker to change its processing of INCLUDE and LIBRARY control statements.

Object files and object libraries from `c89` are passed to the prelinker via INCLUDE and LIBRARY control statements, respectively, in its primary input. Only LIBRARY control statements included in primary input are accepted by the prelinker. Their syntax is:

```
LIBRARY libname
```

where `libname` is a ddname that defines a library. The library can be either an archive file created through the OpenExtensions `ar` utility, or a C370LIB text library with object modules as members. Object libraries from `c89` are passed to the prelinker using such statements. This provides a capability that is best described as named SYSLIBs. These statements are used like SYSLIBs to resolve symbols through autocalls, but they do not all have to be concatenated together as a SYSLIB. BFS files cannot be concatenated.

When the OE option is specified, the prelinker accepts BFS files and CMS files on INCLUDE and LIBRARY control statements.

The OE option causes the prelinker to use POSIX rules for processing its primary input; the order of passed object files and object libraries and their interspersion is significant. The prelinker's primary input is processed sequentially. When a primary input INCLUDE control statement is processed, the prelinker accepts new defined and unresolved symbols occurring in the passed object file. When a primary input LIBRARY control statement is processed, only currently unresolved symbols are
Running under OpenExtensions Services

searched for in the passed object library. A library is processed once only even if it contains definitions of unresolved symbols that are accepted during later processing.

RENAME control statements are processed on output from the prelinker, after all of its input has been processed. Because a library can be processed once only, the SEARCH option on the RENAME control statement has no effect.

Specifying Run-Time Options under OpenExtensions

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

Run-time options needed for the OpenExtensions application program residing in the byte file system can be passed from a #pragma runopts preprocessor directive at compile time. When run-time options are specified in this way, a CEEUOPT control section (CSECT) is created and is linked with the application program by the c89 utility. Because only one CEEUOPT CSECT can be linked with an application program, you should code a #pragma runopts directive in the compilation unit for the main() function.

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

Running under OpenExtensions

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

OpenExtensions Application Program Environments

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

- OpenExtensions shell
- CMS

Placing a CMS Application Program Load Module in the File System

If you have an OpenExtensions C/C++ application program executable file as a CMS native file and want to place it in the BFS, use the following OpenExtensions CMS command to copy the file into a BFS file: openvm putbfs. For a description of this command, see z/VM: OpenExtensions Commands Reference. For examples of using commands to copy CMS files into BFS, see z/VM: OpenExtensions User's Guide.

Running a CMS Module from the OpenExtensions Shell

If your OpenExtensions C/C++ program is a CMS module file on a minidisk or in the shared file system, the only way you can invoke it from the shell is by creating an external link in BFS that points to the file. For example, if you need to execute PROG1 MODULE A, you can create a file in BFS that represents it by using the following command:

openvm create extlink /u/mydir/prog cmsexec prog1 module a
Running under OpenExtensions Services

You can then invoke the module directly from the shell by entering prog assuming that /u/mydir is in the current PATH.

See [z/VM: OpenExtensions Commands Reference](#) for more information on creating external links.

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

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

**Issuing the Executable File Name from the Shell**

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

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

data_crunch

When running such programs, you can specify invocation run-time options only by setting the environment variable _CEE_RUNOPTS before invoking the program. For example, under the shell you can use the following EXPORT command:

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

To further update the run-time options, you need to issue another EXPORT.

**Issuing a Setup Shell Script File Name from the Shell**

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

Basic Building and Running PL/I routines under OpenExtensions

When the run-time option POSIX(ON) is specified, PL/I routines in the Initial Process Thread (IPT) follow the same rules and behave in the same way they do when POSIX(ON) is not in effect.

PL/I routines in non-IPTs, however, must follow the rules described in [z/OS: Language Environment Programming Guide](#). No run-time diagnosis is provided to enforce those rules.
Chapter 4. Initialization and Termination under Language Environment

This chapter describes z/VM considerations when calculating the Language Environment enclave return code. For additional information regarding initialization and termination under Language Environment, see z/OS: Language Environment Programming Guide.

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 and the enclave reason code as follows:

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

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

For information on setting and altering user return codes and calculating the enclave reason code, see z/OS: Language Environment Programming Guide.

z/VM Considerations

The CMS Ready(XXXXX) prompt displays only the last 5 digits of the enclave return code. Under VM, some enclave return codes containing more than 5 digits (for example, 2,000,000 or 3,000,000) are not displayed. In this case, the CMS Ready prompt indicates the following:

Ready(00000);

When a negative number is returned that is greater than 4 digits, only the last 4 digits are displayed. For example, if the return code is -65280, only -5280 is displayed.

You can write a simple REXX EXEC to retrieve the complete enclave return code. The following example extracts the return code and issues a message based on its value:

/* */
'LEMOD' /* Run the Language Environment program */
LE_RC = Rc /* Save the Language Environment enclave return code */
If LE_RC ^= 0 Then Do
   Say 'Nonzero Language Environment enclave return code: ' LE_RC
   Exit 16
Else /* nonzero rc from Language Environment */
   Exit 0
Initialization and Termination
Chapter 5. Using and Handling Messages

This chapter describes z/VM considerations for using Language Environment message services to create, issue, and handle messages for Language Environment-conforming applications.

For more information on using and handling Language Environment messages, see z/OS: Language Environment Programming Guide.

Creating Messages

The following sections explain how to create messages to use in your routines. To create a message, you:
1. Create a message source file
2. Convert the message source file to an ASSEMBLE file with the CEEBLDTX utility
3. Assemble the new message ASSEMBLE file
4. Create a message module table
5. Assign values to message inserts
6. Use messages in code to get message output

Creating a Message Source File

The message source file contains the message text and information associated with each message. Standard tags and format are used for message text and different types of message information. The tags and format of the message source files are used by the CEEBLDTX utility to transform the source file into an ASSEMBLE file.

A file type of SCRIPT is assumed for the source file and the file mode defaults to A. 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 file 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 1 on page 30 shows an example of a message source file with a facility ID of XMP.
Using Messages

:facid.XMP
:msgno.10
:msgsubid.0001
:msgname.EXMPLMSG
:msgclass.I
:msg.This is an example of an insert,
:tab.+1
:ins 1.a simple insert
:msg., within a message.
:xpl.This is a simple example of how to put an insert into a message.
:presp.No programmer response required.
:sysact.No system action is taken.

Figure 1. Example of a Message Source File

For more information on creating Language Environment messages, see z/OS: Language Environment Programming Guide.

Using the CEEBLDCTX Utility

CEEBLDTX is a utility that transforms the message source file into an ASSEMBLE file that can then be assembled and loaded. The syntax of the CEEBLDTX invocation is shown below.

This utility only runs in a regular CMS environment, not in the OpenExtensions Shell & Utilities environment.

```
>> CEEBLDTX in_file out_file options
```

**in_file**

The file name of the SCRIPT file containing the message text source.

**out_file**

The file name of the resulting ASSEMBLE file containing the text version of the messages.

**options**

Can be omitted or be any of the following:

- **C370(filename)**
- **COBOL(filename)**
- **PLI(filename)**
- **BAL(filename)**

COBOL options only:

- **APOST**
- **QUOTE**

APOST/QUOTE specifies which COBOL delimiter to use. APOST is the default.

**filename**

Specifies the name of the file to contain the condition tokens for the messages supplied in in_file in the format requested by the option(s) specified above.

Filename has the following default file types based on the specified language:

- **H** For C
- **COPY** For COBOL
- **COPY** For PL/I
- **COPY** For BAL
Usage Notes:
1. Each parameter is positional. Every parameter, except the options parameter, is required.
2. Under z/VM an equal sign (=) can be substituted for any parameter, except for in_file. Parameters represented by an equal sign (=) are equated with the corresponding parameter previously used.

Files Created by CEEBLDTX

The CEEBLDTX utility creates several files. The ASSEMBLE file can be assembled into a loadable text file. When the name of this file is placed in a message module table, the Language Environment message services can dynamically access the file. See z/OS: Language Environment Programming Guide for more information about creating a message module table.

The COPY or INCLUDE file contains the declarations for the condition tokens associated with each message in the message source file. When this file is included in the source routine, the condition tokens can be used to reference the message. The :msgname. tag indicates the symbolic name of the condition token.

To use the CEEBLDTX utility with the sample file shown in Figure 1 on page 30 you would issue:

CEEBLDTX example exmplasm pli(exmplcop)

The in_file is EXAMPLE SCRIPT, the out_file is EXMPLASM ASSEMBLE, and the PL/I COPY file is EXMPLCOP COPY.

Use High Level Assembler to assemble the ASSEMBLE file into a loadable TEXT file; for example, on z/VM you can issue the command:

HLASM exmplasm

CEEBLDTX Error Messages

The following is a list of Language Environment CEEBLDTX errors. For more information on CEEBLDTX errors, see z/OS: Language Environment Programming Guide.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Error Message</th>
<th>Explanation</th>
<th>Programmer response</th>
</tr>
</thead>
<tbody>
<tr>
<td>0004</td>
<td>syntax error</td>
<td>The command entered contained a syntax error.</td>
<td>Correct the syntax error and reissue the command.</td>
</tr>
<tr>
<td>0028</td>
<td>ssssssss SCRIPT not found on any accessed disk.</td>
<td>The SCRIPT file with the name ssssssss does not exist.</td>
<td>Make sure the name is given correctly and is accessible.</td>
</tr>
<tr>
<td>0036</td>
<td>Disk A must be accessed as Read/Write.</td>
<td>On VM, the A-disk must be writable to write the outfile files.</td>
<td>Specify an A-disk that is write accessible.</td>
</tr>
<tr>
<td>0040</td>
<td>Error on line nnn in message nnnn - insert number greater than mmmm.</td>
<td>An insert number greater than the allowable maximum was specified. The current maximum allowable insert number is 9.</td>
<td>Specify an insert number of 9 or less.</td>
</tr>
<tr>
<td>0044</td>
<td>Error on line nnn - Duplicate :FACID. tags found within the given script file.</td>
<td>Only one facility ID can be specified in the SCRIPT file.</td>
<td></td>
</tr>
</tbody>
</table>

Using Messages

Chapter 5. Using and Handling Messages  31
Using Messages

Programmer response: Specify only one facility ID in the SCRIPT file.

Return Code=0048  No :FACID. found within the given script file.

Explanation: A 3-character facility ID must be specified in the SCRIPT file with the :facid. 31 tag.

Programmer response: Specify a 3-character facility ID with the :facid. 33 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  Error on line nnn in message mmmm - number of hex digits not divisible by 2

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  Error on line nnn in message mmmm - invalid hexadecimal digits

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  Error on line nnn in message mmmm - number of DBCS bytes not divisible by 2

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  ASSEMBLE out_file name must be longer than the message facid 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  Error on line nnn - message facid pppp 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  Error on line nnn - message class class is 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  Error on line nnn - tag not recognized

Explanation: A tag that was not recognized was encountered.

Programmer response: Check the tag for proper spelling and use.

Return Code=0084  Error on line nnn - first tag not :FACID.

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  Error on line nnn - unexpected tag.

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  Error on line nnn in message mmmm - duplicate errortag tags

Explanation: Duplicate :msgname., :msgclass., or :msgsubid. tags were found for a single message.

Programmer response: Remove the extra tag from the message script.

Return Code=0096  No :MSGNO. tags found within the given script file.

Explanation: A message file must have at least one message in it, and it must be denoted by a :msgno. tag.

Programmer response: Specify at least one message in the message file.
Using Messages

Run-Time Messages with POSIX

When your C application is running with POSIX(ON), some messages have changed both facility ID and message number. Messages that had a facility ID of EDC and ranged from message number 6000 through 6008 prior to running with POSIX(ON) now have a facility ID of CEE and use message numbers 5201 through 5209. Messages 5210 through 5233 are new for POSIX(ON) and thus do not have a corresponding POSIX(OFF) message number, except for message 5223, which has a facility ID of EDC and a message number of 6008 while running with POSIX(OFF). When your C application is running with POSIX(OFF), facility ID EDC is still used for message numbers 6000 through 6009.

If your C application is coded to respond to specific facility IDs or specific message numbers for processing, you must specify POSIX(OFF) to receive the facility ID of EDC and message numbers 6000 through 6009.

Table 4 shows the conditions, their condition numbers, and facility IDs.

Table 4. Condition Tokens with POSIX

<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>SIGINT</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>
</tbody>
</table>
Run-Time Messages with POSIX

Table 4. 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>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>
<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>SIGTTIN</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>
</tbody>
</table>

Handling Message Output

The following sections provide information about directing message output and displaying messages under Language Environment, C, C++, COBOL, 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

Run-time messages are directed to a common Language Environment message file. You can use the MSGFILE run-time 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.

Table 5 lists the SYSOUT definitions and MSGFILE default attributes for CMS:

Table 5. 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>CMS</td>
<td>FILEDEF SYSOUT TERMINAL</td>
<td>LRECL 121, BLKSIZE 121, RECFM FBA, NOCHANGE</td>
</tr>
</tbody>
</table>

Notes:

1. When output is directed to the terminal, ASA control characters are replaced by a single space.
When you direct run-time messages to an I/O device, the method you should use also depends on the operating system. Table 6 lists methods for directing run-time messages to an I/O device under CMS and provides references for additional information on this topic.

Table 6. 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 additional information, see:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMS</td>
<td>Use a FILEDEF statement to define a ddname for a file.</td>
<td>&quot;Using FILEDEF to Define Input and Output Files&quot; on page 14</td>
</tr>
</tbody>
</table>

**Note:** You can specify the same message file across nested enclaves.

**Using MSGFILE under OpenExtensions**

If your application is running under the OpenExtensions shell or any environment that has file descriptor 2 (FD2) open, MSGFILE output is directed to the FD2 print destination. Under the shell, this is typically your terminal. If FD2 is closed when your application is invoked (via spawn() or exec()), no message file is created.

For dump services, 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 20030325 for March 25, 2003).
- **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.

**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.

Run-time 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 CMS interactive, stdout defaults to the terminal.

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 7 on page 36 lists the types of C/C++ output, the types of messages associated with them, and the destination of the message output.
Handling Message Output

Table 7. C and C++ 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 messages (CEExxx)</td>
<td>Language Environment unhandled conditions</td>
<td>MSGFILE ddname</td>
</tr>
<tr>
<td>MSGFILE output</td>
<td>C library messages</td>
<td>C/C++ unhandled conditions (EDCxxxx)</td>
<td>MSGFILE ddname</td>
</tr>
<tr>
<td>stderr messages</td>
<td>perror() messages (EDCxxx)</td>
<td>Issued by a call to perror()</td>
<td>MSGFILE ddname</td>
</tr>
<tr>
<td>stderr messages</td>
<td>User output sent explicitly to stderr</td>
<td>Issued by a call to fprintf()</td>
<td>MSGFILE ddname</td>
</tr>
<tr>
<td>stdout messages</td>
<td>User output sent explicitly to stdout</td>
<td>Issued by a call to fprintf()</td>
<td>stdout</td>
</tr>
</tbody>
</table>

You can control the destination of stderr and stdout output by using the Language Environment MSGFILE run-time option, the C freopen() function, or by invoking redirection services at run time.

Table 8 lists the possible destinations of redirected stderr and stdout standard stream output.

Table 8. C/C++ Redirected Stream Output

<table>
<thead>
<tr>
<th>stderr not redirected</th>
<th>stdout not redirected</th>
<th>stdout redirected to other than stdout</th>
<th>stdout redirected to stderr</th>
</tr>
</thead>
<tbody>
<tr>
<td>stderr redirected to destination other than stdout</td>
<td>stdout redirected to stderr</td>
<td>Both to stdout</td>
<td>Both to stdout</td>
</tr>
<tr>
<td>stderr to MSGFILE</td>
<td>stdout to itself</td>
<td>stderr to its other destination</td>
<td>Both to the other stdout destination</td>
</tr>
<tr>
<td>stdout redirected to stderr</td>
<td>stdout redirected to other than stdout</td>
<td>Both to MSGFILE</td>
<td>Both to the other stderr destination</td>
</tr>
</tbody>
</table>

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 XL C/C++ for z/VM: User's Guide.

Using COBOL I/O Statements

Language Environment manages all COBOL output directed to the system-logical output device. This includes output from:

- DISPLAY ... UPON SYSOUT
- READY TRACE (OS/VS COBOL only)
- EXHIBIT (OS/VS COBOL only)
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 CMS file to which the run-time messages are written depends on the combination of ddnames specified in the OUTDD compiler option and the MSGFILE run-time option.

If the ddname in OUTDD matches the ddname specified in the MSGFILE run-time option, the output is synchronized with the run-time messages and placed in the CMS file designated by the MSGFILE run-time option.

If the ddname in OUTDD does not match the ddname specified in the MSGFILE run-time 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 5 on page 34.

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 5 on page 34.

The possible ddname specification combinations for OUTDD and MSGFILE and the locations where display output and run-time messages are routed are summarized in Table 9 on page 38.
### Handling Message Output

**Table 9. Run-time Message and DISPLAY Destinations for OUTDD and MSGFILE ddname Specifications under VM**

<table>
<thead>
<tr>
<th>ddname Speciation</th>
<th>FILEDEFs Issued?</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSGFILE(SYSOUT)</td>
<td></td>
<td>Messages and DISPLAY data are routed to the destination defined for SYSOUT.</td>
</tr>
<tr>
<td>OUTDD(SYSOUT)</td>
<td>Yes, for SYSOUT</td>
<td>Language Environment dynamically allocates FILEDEF SYSOUT TERM for messages and DISPLAY data.</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Language Environment dynamically allocates FILEDEF SYSOUT TERM, the message destination.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Language Environment dynamically allocates ddname to FILEDEF ddname DISK FILE ddname A1, the DISPLAY data destination.</td>
</tr>
</tbody>
</table>

For more information about directing COBOL output, refer to *COBOL for OS/390 & VM Programming Guide* or *COBOL for MVS & VM Programming Guide*.

### Using PL/I I/O Statements

Run-time messages in PL/I routines are directed to the file specified by the Language Environment MSGFILE run-time 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 run-time 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 or FILEDEF are used.
Handling Message Output

- If SYSPRINT DD or FILEDEF is not present at first file reference, Language Environment dynamically allocates a file with IBM-supplied attributes. See Table 5 on page 34 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. 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.

Chapter 6. Using Run-Time 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 chapter provides z/VM specific information about using these run-time user exits. For more information on using run-time user exits, see z/OS Language Environment Programming Guide.

Understanding the Basics

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 run-time options that establish characteristics of the environment. This is done prior to 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 the sections User Exits Supported under Language Environment and When User Exits Are Invoked on page 43, 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, the z/OS Language Environment Programming Guide describes 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 (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 10.

Table 10. User Exits Supported under Language Environment

<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), PL/I or Language</td>
<td>Enclave initialization</td>
</tr>
<tr>
<td></td>
<td>Environment-conforming assembler.</td>
<td>Enclave termination</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. When CEEBXITA is linked in your load module, it
Run-Time User Exits

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.

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 a CMS INCLUDE command (see "Using the INCLUDE Command" on page 12 for more information). Any time that the application-specific exit is modified, it must be relinked with the application.

For a description of the assembler user exit interface and the HLL user exit interface, see z/OS: Language Environment Programming Guide.

PL/I and C Compatibility

The following OS PL/I Version 2 Release 3 user exit is supported for compatibility under Language Environment:

- IBMBXITA (VM version)

For information about IBMBXITA and IBMBINT, see PL/I for MVS & VM Compiler and Run-Time Migration Guide and 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 11 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 OS/390 Batch or VM, 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>IBMFXITA</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 12. Sample Assembler User Exits for 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>CEEBXITB</td>
<td>z/VM (default)</td>
<td>CEEBXITB ORIGINAL</td>
<td></td>
</tr>
</tbody>
</table>
If you install Language Environment at your site without modifying it, your system default is CEEBXITB. You can find the source code for CEEBXITB on the z/VM disk where Language Environment is installed with the name CEEBXITB ORIGINAL.

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 run-time support

**When User Exits Are Invoked**

Figure 2 shows the timing of the invocations of the user exits at initialization and termination processing.

In Figure 2, run-time user exits are invoked in the following sequence:
1. Assembler user exit is invoked for enclave initialization.
Run-Time User Exits

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 prior to 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 sections.

**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 run-time environment. For example, in the assembler user exit you can specify the stack and heap run-time options and allocate data sets. You can also use the user exit to interrogate program parameters and change them if you want. In addition, you can specify run-time options in the user exit by using the CEEAUE_A_OPTIONS field of the assembler interface.

**z/VM Considerations**

The behavior of the IBM-supplied version of CEEBXITA differs, depending upon whether you are running your application under z/VM or OS/390.

- Under OS/390, CEEBXITA returns control to Language Environment initialization.
- VM only — CEEBXITA issues FILEDEFs for ddnames CEEDUMP, SYSOUT, and SYSIN, then returns control to Language Environment initialization.

**Note for C-Specific Installations**

This set of FILEDEFs differs from the ones in IBMBXITA that the pre-AD/Cycle version of C used.

**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).
- The exit must be relinked with Language Environment initialization/termination routines after modification.

If a user exit is modified, you are responsible for conforming to the interface shown in Figure 3 on page 45. Note that this user exit must be written in assembler. 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 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.
2. Termination of the first enclave within a process.
5. 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:

### Figure 3. Interface for CEEBXITA Assembler User Exit

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:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Fullword Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CEEAUE_LEN</td>
</tr>
<tr>
<td>4</td>
<td>CEEAUE_FUNC</td>
</tr>
<tr>
<td>8</td>
<td>CEEAUE RETC</td>
</tr>
<tr>
<td>12</td>
<td>CEEAUE_RSNC</td>
</tr>
<tr>
<td>16</td>
<td>CEEAUE_FLAGS</td>
</tr>
<tr>
<td>20</td>
<td>CEEAUE_A_CC_PLIST</td>
</tr>
<tr>
<td>24</td>
<td>CEEAUE_WORK</td>
</tr>
<tr>
<td>28</td>
<td>CEEAUE_A_OPTIONS</td>
</tr>
<tr>
<td>32</td>
<td>CEEAUE_USERWD</td>
</tr>
<tr>
<td>36</td>
<td>CEEAUE_A_AB_CODES</td>
</tr>
<tr>
<td>40</td>
<td>CEEAUE_FBCODE</td>
</tr>
<tr>
<td>44</td>
<td>CEEAUE_PAGE</td>
</tr>
</tbody>
</table>
Run-Time User Exits

- 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.)

**CEEAE_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.

**CEEAE_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 4:

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>CEEAE_ABTERM</th>
<th>0... Normal termination</th>
<th>1... Abnormal termination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CEEAE_ABND</td>
<td>0... Terminate with CEEAUE_RETC</td>
<td>1... ABEND with CEEAUE_RETC and CEEAUE_RSNC given</td>
</tr>
<tr>
<td></td>
<td>CEEAE_DUMP</td>
<td>.0... If CEEAE_ABND=0, ABEND with no dump</td>
<td>.1... If CEEAE_ABND=1, ABEND with a dump</td>
</tr>
<tr>
<td></td>
<td>CEEAE_STEPS</td>
<td>...0... ABEND the task</td>
<td>...1... ABEND the step</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...0000 Reserved (must be zero)</td>
<td></td>
</tr>
</tbody>
</table>

**Byte 1** 0000 0000 Reserved for future use
**Byte 2** 0000 0000 Reserved for future use
**Byte 3** 0000 0000 Reserved for future use

*Figure 4. CEEAE_FLAGS Format*

Byte 0 (CEEAE_FLAG1) has the following meaning:

**CEEAE_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.

**CEEAE_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 run-time option does not affect the setting of CEEAUE_ABND.

When the ABTERMENC(ABEND) run-time 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. For a detailed explanation of how the CEEAUE_ABND parameter can affect the behavior of the ABTERMENC run-time option, see [z/OS: Language Environment Programming Guide](#).

**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.

z/VM currently honors the dump request on an abend if you specify the destination in one of the following FILEDEF statements:

- FILEDEF SYSABEND PRINTER
- FILEDEF SYSUDUMP PRINTER

**CEEAUE_STEPS** (output parameter)

- **OFF** Indicates that when you request an abend, an abend is issued to abend the entire TASK.
- **ON** This parameter is ignored under z/VM.

**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_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 run-time options. These options are honored only during the initialization of an enclave. When invoked for enclave termination, this field is ignored.

These run-time options override all other sources of run-time options except those that are specified as NONOVR in the installation default run-time options.

The LIBRARY and VERSION run-time options cannot be specified in the CEEAUE_A_OPTIONS output string. When the assembler user exit is invoked, it is too late to change any of these options.

**CEEAUE_USERWD** (input/output parameter)

A fullword whose value is maintained without alteration and passed to every
Run-Time User Exits

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.

**CEEUAUE_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 prior to 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'.

**CEEUAUE_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.

**CEEUAUE_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 4K.

The IBM-supplied default setting for CEEUAUE_PAGE is 32768 (32K).

If CEEUAUE_PAGE is set to zero, PL/I BASED variables can be placed on other than 4K-page boundaries.

CEEUAUE_PAGE is honored only during enclave initialization, that is, when CEEUAUE_FUNC is 1 or 3.

The offset of CEEUAUE_PAGE under Language Environment is different than under OS PL/I Version 2 Release 3.
Chapter 7. 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.

This chapter describes z/VM specific considerations for Language Environment preinitialization service routines. For more information on preinitialization, see z/OS Language Environment Programming Guide.

Restriction: XPLINK programs are not supported in the PreInit environment.

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 5.

```
SERV_RTNS  Count
User Word
@WorkArea
@LOAD
@DELETE
@GETSTORE
@FREESTORE
reserved
@EXCEPRTN
@MSGRTN
```

Figure 5. 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 CEEPPIP(init_main) and CEEPPIP(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 5.

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
Preinitialization Services

@LOAD and @DELETE service routines. If you specify the @GETSTORE and @FREESTORE service routines, you must also specify the @LOAD and @DELETE service 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 5 on page 49 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.

**@LOAD**
This routine loads named routines for application management. Under VM, this routine can load modules from nucleus extension, saved segment, or relocatable load library members. The search sequence is in the same order. 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 13.

<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>4</td>
<td>Successful — found as a CMS nucleus extension</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>Successful — loaded as a CMS shared segment</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>
</tbody>
</table>
Table 13. Return and Reason Codes (continued)

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Reason Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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. Under VM, this routine can load modules from nucleus extension, saved segment, or relocatable load modules. The search sequence is in the same order. 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 14.

Table 14. Return and Reason Codes

<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).
Preinitialization Services

Bit zero in Flags is ON if the storage is required below the 16M line. The remaining bits are reserved for future use and must be zero. Bit zero in Flags is OFF if the storage required can be allocated anywhere.

**Stg_address**
The fullword address of the storage 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 15.

<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 16.

<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>

@EXCEPTRN
This routine traps program interruptions and abends for condition management. The parameter list passed contains the following:
Preinitialization Services

**Handler_addr**
During an initialization call, this parameter contains the address of the CEL 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). (Note that this token is different from the PIPI environment token used with CEEPPIPI 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, if the TRAP option is in effect the common library puts the address of the Language Environment exception in the first field of the above parameter list, and sets the environment token field to a value that will be passed on to the exception 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:

```plaintext
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:

```plaintext
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);
```
Preinitialization Services

The return and reason codes that the exception router must use are listed in Table 17.

Table 17. Return and Reason Codes

<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>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 router must determine if the established exception handler is interested in the exception (by examining abend and check flags). If the exception handler is not interested in the exception, the exception router must treat the program as in error, but can assume the environment for the thread to be functional and reusable. If the exception handler is interested in the exception, the exception router must invoke the exception handler, passing the parameters listed in Table 18.

Table 18. Parameters for EXCEPTRTN

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Attributes</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment Token</td>
<td>Pointer</td>
<td>Input</td>
</tr>
<tr>
<td>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 exception handler are listed in Table 19.

Table 19. Return and Reason Codes

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Reason Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0           | 0           | 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 router can assume the environment for the thread to be functional and reusable. |
| 0           | 4           | 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. |
| 4           | 0           | Resume execution using the updated SDWA.  
  The invoked exception handler will have already used the SETRP RTM macro to set the SDWA for correct resumption. |

During termination, the exception router is invoked with the exception handler address (first parameter) set to zero to de-establish the exit (if it was established during initialization).

When a nested enclave is created, the Language Environment exception handler 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 exception 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 router 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 router should not invoke the Language Environment exception handler if active I/O has been halted and is not restorable.
- This service requires an XA or ESA environment.
- This service is not supported under CMS.

If an exception occurs while the exception handler is in control before another exception handler exit has been stacked, the exception router should assume that the exception could not be handled and that the environment for the program (thread) is damaged. In this case, the exception router should force termination of the preinitialized environment.

**@MSGRTN**

This routine allows error messages to be processed by the caller of the application.

If the message pointer is zero, your message routine is expected to return the size of the line to which messages are written (in the line_length field). This allows messages to be formatted correctly — that is, broken at places such as blanks.

**Message**

A pointer to the first byte of text that is printed, or zero (input parameter).

**Msg_len**

The fixed binary(31) length of the message (input parameter).

**User word**

A 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 20 (output parameters).

---

**Table 20. Return and Reason Codes**

<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

In the example shown in Figure 6 on page 57, assembler program ASMPIPI ASSEMBLE invokes CEEPIPI to:

- Initialize a subroutine environment under Language Environment
- Load and call a reentrant HLL subroutine
- Terminate the Language Environment environment

For examples of the program HLLPIPI written in C, COBOL, and PL/I, see z/OS Language Environment Programming Guide.

When using HLLPIPI C under z/VM, enter these commands:

LOAD HLLPIPI (RLDSAVE RESET HLLPIPI
GENMOD HLLPIPI
Preinitialization Services

Figure 6. Assembler Driver That Creates a Preinitialized Environment (Part 1 of 3)
Preinitialization Services

WTO 'ASMPIPI : call to CEEPIPI(INIT_SUB) failed',ROUTCDE=11
C R2,=F'8' Check for partial initialization
BE TSUB Yes.. go do PIPI termination
* No.. issue message & quit
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 *
  L R15,PPRTNPTR Get address of CEEPIPI routine
  CALL (15),(CALLSUB,PTBINDEX,TOKEN,PARMPTR,
  SUBRETC,SUBRSNC,SUBFBC) Invoke CEEPIPI routine
*
  Check return code:
  LTR R2,R15 Is R15 = zero?
  BZ TSUB Yes (success).. go to next section
  BZ DONE No (failure).. issue message & quit
WTO 'ASMPIPI : call to CEEPIPI(CALL_SUB) failed',ROUTCDE=11
ABEND (R2),DUMP Abend with bad RC and dump memory
*
* Terminate the environment
* TSUB EQU *
  L R15,PPRTNPTR Get address of CEEPIPI routine
  CALL (15),(TERM,TOKEN,ENV_RC) Invoke CEEPIPI routine
*
  Check return code:
  LTR R2,R15 Is R15 = zero?
  BZ DONE Yes (success).. go to next section
  BZ DONE No (failure).. issue message & quit
WTO 'ASMPIPI : call to CEEPIPI(TERM) failed',ROUTCDE=11
ABEND (R2),DUMP Abend with bad RC and dump memory
*
* Standard exit code.
* DONE EQU *
  LA R15,0 Passed return code for system
  L R13,SAVE+4 Get address of caller's save area
  L R14,12(R13) Reload caller's register 14
  LM R0,R12,20(R13) Reload caller's registers 0-12
  BR R14 Branch back to caller
*
* =====================================================================
* CONSTANTS and SAVE AREA.
* =====================================================================
SAVE DC 18F'0'
PPRTNPTR DS A Save the address of CEEPIPI routine
*
* Parameters passed to a CEEPIPI(INIT_SUB) call.
* INITSUB DC F'3' Function code to initialize for subr
@CEXPTBL DC A(PPTBL) Address of PIP Table
@SRVRTNS DC A(0) Addr of service-rtns vector, 0 = none
RUNMPTO DC CL255' ' Fixed length string of runtime optns
TOKEN DS F Unique value returned (output)
*
* Parameters passed to a CEEPIPI(CALL_SUB) call.
* CALLSUB DC F'4' Function code to call subroutine
PTBINDEX DC F'0' The row number of PIP Table entry
PARMPTR DC A(0) Pointer to @PARMLIST or zero if none
SUBRETC DS F Subroutine return code (output)
SUBRSNC DS F Subroutine reason code (output)
SUBFBC DS 3F Subroutine feedback token (output)
*

Figure 6. Assembler Driver That Creates a Preinitialized Environment (Part 2 of 3)
Preinitialization Services

* Parameters passed to a CEEPIPI(TERM) call.
  *
  TERM DC F'5'           Function code to terminate
  ENV_RC DS F            Environment return code (output)
  *
  * =====================================================================
  * PIPI Table.
  * =====================================================================
  PPTBL CEEXPIT ,       PIPI Table with index
    CEEXPITY HLLPIPI,0   0 = dynamically loaded routine
  *
    CEEXPITS ,           End of PIPI table
  *
    LTORG
    R0 EQU 0
    R1 EQU 1
    R2 EQU 2
    R3 EQU 3
    R4 EQU 4
    R5 EQU 5
    R6 EQU 6
    R7 EQU 7
    R8 EQU 8
    R9 EQU 9
    R10 EQU 10
    R11 EQU 11
    R12 EQU 12
    R13 EQU 13
    R14 EQU 14
    R15 EQU 15
    END ASMPIPI

Figure 6. Assembler Driver That Creates a Preinitialized Environment (Part 3 of 3)
Chapter 8. Using Nested Enclaves

An enclave is a logical run-time structure that supports the execution of a collection of routines (for a detailed description of Language Environment enclaves, see [z/OS Language Environment Programming Guide]).

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 chapter, 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).

Understanding the Basics

In Language Environment, you can use the following methods to create a child enclave:

- the SVC LINK or CMSCALL commands (for more information about SVC LINK and CMSCALL, see your system reference)
- the C `system()` function (for more information about `system()`, see [z/OS XL C/C++ Programming Guide])
- the PL/I FETCH and CALL to any of the following PL/I routines with PROC OPTIONS(MAIN) specified:
  - 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 book, 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 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 run-time option forced ON).
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 run-time options
- The default condition handling behavior of each type of child enclave
- The setting of the TRAP run-time 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 (SVC LINK, CMSCALL, C system() function, and PL/I FETCH and CALL of a fetchable main) will behave.

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, PL/I, or Language Environment-conforming assembler (generated by use of the CEEENTRY and associated macros).

How Run-Time Options Affect Child Enclaves

Run-time 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 or CMSCALL

The behavior of a child enclave created by an SVC LINK or CMSCALL is determined by the language of its main routine: C, C++, COBOL, PL/I, or Language Environment-conforming assembler (generated by use of the CEEENTRY and associated macros).

If you want to issue a LINK to a routine, you must first either use the LKED command to put the target routine's object module into a LOADLIB or use the LOAD command with the RLDSAVE option and the GENMOD command with the NOMAP option to create a relocatable load module. For more information about the LKED command, see "Link-Editing with the LKED Command" on page 14.

If you want to issue a CMSCALL to a routine, you must first use either the LOAD and GENMOD commands or the BIND command to put the target routine's object code into a CMS MODULE. For more information about these commands see "Using the LOAD and INCLUDE Commands" on page 9, "Using the GENMOD Command" on page 12, and "Using the BIND Command" on page 13.

How Run-Time Options Affect Child Enclaves

Child enclaves created by an SVC LINK or CMSCALL get run-time options differently, depending on the language that the main routine of the child enclave is written in.
Using Nested Enclaves

Child Enclave Has a C, C++, PL/I, or Language Environment-Conforming Assembler Main Routine: If the main routine of the child enclave is written in C, C++, PL/I, or in Language Environment-conforming assembler, the child enclave gets its run-time options through a merge from the usual sources (see Z/OS Language Environment Programming Guide for more information). Therefore, you can set run-time options on an enclave-by-enclave basis.

Child Enclave Has a COBOL Main Program: If the main program of the child enclave is written in COBOL, the child enclave inherits the run-time options of the creating enclave. Therefore, you cannot set run-time 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 or CMSCALL, regardless of the language of its main, the entire process is terminated.

Condition handling in child enclaves created by SVC LINK or CMSCALL varies, depending on the language of the child's main routine, the setting of the TRAP run-time 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 21. Handling Conditions in Child Enclaves

<table>
<thead>
<tr>
<th>If the Child Enclave Was Created By:</th>
<th>See:</th>
</tr>
</thead>
<tbody>
<tr>
<td>An SVC LINK or CMSCALL under CMS and has a C or Language Environment-conforming assembler main routine</td>
<td>Table 22</td>
</tr>
<tr>
<td>An SVC LINK or CMSCALL under CMS and has a COBOL main program</td>
<td>Table 23 on page 64</td>
</tr>
<tr>
<td>An SVC LINK or CMSCALL under CMS and has a PL/I main routine</td>
<td>Table 24 on page 65</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 22 shows the unhandled condition behavior under CMS.

Table 22. Unhandled Condition Behavior in a C or Assembler Child Enclave, under CMS

<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 22. Unhandled Condition Behavior in a C or Assembler Child Enclave, under CMS (continued)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent Enclave TRAP(ON)</th>
<th>Child Enclave TRAP(ON)</th>
<th>Parent Enclave TRAP(OFF)</th>
<th>Child Enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Language Environment abend</td>
<td>Process terminated with original abend code</td>
<td>Process terminated with original abend code</td>
<td>Process terminated with original abend code</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 CMS message</td>
</tr>
</tbody>
</table>

Child Enclave Has a COBOL Main Program: Child enclaves created by SVC LINK or CMSCALL that have a COBOL main program inherit the run-time options of the parent enclave that created them. Therefore, the TRAP setting of the parent and child enclaves is always the same.

Table 23 shows unhandled condition behavior under z/VM.

Table 23. Unhandled Condition Behavior in a COBOL Child Enclave, under z/VM

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent Enclave TRAP(ON)</th>
<th>Parent Enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Enclave TRAP(ON)</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>
</tr>
<tr>
<td>Unhandled condition severity 2 or above</td>
<td>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with abend U4094 RC=40</td>
</tr>
<tr>
<td>Non-Language Environment abend</td>
<td>Process terminated with original abend code</td>
<td>Process terminated with original abend code</td>
</tr>
<tr>
<td>Program check</td>
<td>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with CMS message</td>
</tr>
</tbody>
</table>

Child Enclave Has a PL/I Main Routine: Table 24 on page 65 lists unhandled condition behavior under z/VM.
Using Nested Enclaves

Table 24. Unhandled Condition Behavior in a PL/I Child Enclave, under z/VM

<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>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>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with abend U4094 RC=40</td>
</tr>
<tr>
<td>Non-Language Environment abend</td>
<td>Process terminated with original abend code</td>
<td>Process terminated with original abend code</td>
<td>Process terminated with original abend code</td>
<td>Process terminated with original abend code</td>
</tr>
<tr>
<td>Program check</td>
<td>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with abend U4094 RC=40</td>
<td>Process terminated with CMS message</td>
</tr>
</tbody>
</table>

Creating Child Enclaves Using the C system() Function

Child enclaves created by the C system() function get run-time options through a merge from the usual sources (for more information, see [z/OS: Language Environment Programming Guide](https://www.ibm.com/support/knowledgecenter/s9827a_2.4.2/info/sbcocna/bxensenv.htm)). Therefore, you can set run-time options on an enclave-by-enclave basis. For information on the system() function when running with POSIX(ON), see [XL C/C++ for z/VM: Runtime Library Reference](https://www.vm.ibm.com/zvm/zterminal.html).

Run-time options specified in the PARM= portion of the system() function are ignored when you perform a system() function to a COBOL program in the following form:

```c
system("PGM=program_name,PARM='...'")
```

However, run-time options are merged from CEEDOPT, CEEUOPT, and the CEEAUE_A_OPTIONS from the assembler user exit.

OpenExtensions Considerations

To create a nested enclave under Open Extensions, 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 run-time 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 25 describes the effects of TRAP(ON|OFF) for enclaves that are created by the system() function on a z/VM system.

Table 25. Unhandled Condition Behavior in a system()-Created Child Enclave, under z/VM

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent Enclave TRAP(ON)</th>
<th>Parent Enclave TRAP(OFF)</th>
<th>Child Enclave TRAP(ON)</th>
<th>Child Enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<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>Process terminated with original abend code</td>
<td>Process terminated with original abend code</td>
<td>Process terminated with original abend code</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 CMS message</td>
</tr>
</tbody>
</table>

Creating Child Enclaves Containing a PL/I Fetchable Main

Under z/VM, the target load module can only be a member of a LOADLIB or be in a saved segment or relocatable load module. The target load module cannot be on a text deck or be a member of a TXTLIB.

Additional fetch and call considerations of PL/I fetchable mains are discussed in "Special Fetch and Call Considerations" on page 67.

How Run-Time Options Affect Child Enclaves

Child enclaves created when you issue a FETCH and CALL of a fetchable main get run-time options through a merge from the usual sources (see z/OS: Language Environment Programming Guide for more information). Therefore, you can set run-time 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 run-time option are in the parent and child enclave, the following might cause the child enclave to terminate:

- Unhandled user abend
- Unhandled program check

Table 26 describes the unhandled condition behavior in a child enclave that is created under z/VM.

Table 26. Unhandled Condition Behavior in a Child Enclave That Contains a Fetchable Main, under z/VM

<table>
<thead>
<tr>
<th>Condition</th>
<th>Parent Enclave TRAP(ON)</th>
<th>Parent Enclave TRAP(OFF)</th>
<th>Child Enclave TRAP(ON)</th>
<th>Child Enclave TRAP(OFF)</th>
</tr>
</thead>
<tbody>
<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>Process terminated with original abend code</td>
<td>Process terminated with original abend code</td>
<td>Process terminated with original abend code</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 U4036 RC=2</td>
<td>Resume parent enclave, and ignore condition</td>
<td>Process terminated with CMS message</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 [z/OS: Language Environment Programming Guide](#) 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.
Using Nested Enclaves

- 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 returns for each type of child enclave (for more information about the CEE3PRM callable service, see z/OS: Language Environment Programming Reference)
- 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

CEE3PRM returns to the calling routine the user parameter string that was specified at program invocation. Only program arguments are returned.

See Table 27 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 28 on page 69) and returned to you.

Table 27. Determining the Command-Line Equivalent

<table>
<thead>
<tr>
<th>Language</th>
<th>Option</th>
<th>Suboption</th>
<th>CMSCALL or SVC LINK on z/VM</th>
<th>system() on z/VM</th>
<th>FETCH/CALL of a PL/I main</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>#pragma runopts(PLIST)</td>
<td>HOST, CMS, MVS</td>
<td>CMS extended argument list (R0)</td>
<td>PARM=, or the parameter string from the command string passed to system()</td>
<td>Not allowed</td>
</tr>
<tr>
<td>C++</td>
<td>PLIST and TARGET compiler options</td>
<td>Default</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td>COBOL</td>
<td>N/A</td>
<td>CMS extended argument list (R0)</td>
<td>CMS extended argument list (R0)</td>
<td>Null</td>
<td></td>
</tr>
</tbody>
</table>
Using Nested Enclaves

Table 27. Determining the Command-Line Equivalent (continued)

<table>
<thead>
<tr>
<th>Language</th>
<th>Option</th>
<th>Suboption</th>
<th>CMSCALL or SVC LINK on z/VM</th>
<th>system() on z/VM</th>
<th>FETCH/CALL of a PL/I main</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/I</td>
<td>SYSTEM compiler option</td>
<td>MVS</td>
<td>Halfword length-prefixed string pointed to R1</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>CMS</td>
<td>CMS extended argument list</td>
<td>PARM= or the parameter string from the command string passed to system()</td>
<td>User parameters passed through CALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CICS, CMSTPL, 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, CMS, MVS</td>
<td>CMS extended argument list (R0)</td>
<td>PARM=, or the parameter string from the command string passed to system()</td>
<td>Not allowed</td>
</tr>
</tbody>
</table>

If Table 27 on page 68 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 28. The command-line equivalent depends on the language of your routine and the run-time options specified for it.

Table 28. Determining the Order of Run-Time Options and Program Arguments

<table>
<thead>
<tr>
<th>Language of Routine</th>
<th>Run-Time Options in Effect?</th>
<th>Order of Run-Time Options and Program Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>#pragma runopts(EXECOPS)</td>
<td>run-time options / user parms</td>
</tr>
<tr>
<td></td>
<td>#pragma runopts(NOEXECOPS)</td>
<td>entire string is user parms</td>
</tr>
<tr>
<td>C++</td>
<td>Compiled with EXECOPS (default)</td>
<td>run-time 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 / run-time options</td>
</tr>
<tr>
<td></td>
<td>CBLOPTS(OFF)</td>
<td>run-time options / user parms</td>
</tr>
<tr>
<td>PL/I</td>
<td>PROC OPTIONS(NOEXECOPS) or SYSTEM(CICS</td>
<td>run-time options / user parms</td>
</tr>
<tr>
<td></td>
<td>SYSTEM(IMS</td>
<td>TSO) is not specified.</td>
</tr>
<tr>
<td></td>
<td>PROC OPTIONS(NOEXECOPS) is specified, or NOEXECOPS is not specified but SYSTEM (CICS</td>
<td>entire string is user parms</td>
</tr>
<tr>
<td></td>
<td>IMS</td>
<td>TSO) is. See PL/I Main Procedure Parameter Passing Considerations on page 113 for more information on the SYSTEM compile option.</td>
</tr>
<tr>
<td>Language Environment-conforming assembler</td>
<td>CEENTRY EXECOPS=ON</td>
<td>run-time options / user parms</td>
</tr>
<tr>
<td></td>
<td>CEENTRY EXECOPS=OFF</td>
<td>entire string is user parms</td>
</tr>
</tbody>
</table>
Using Nested Enclaves

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:

- SVC LINK or CMSCALL 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 “How the Language Environment Enclave Return Code Is Calculated” on page 27 for more information.) The reason code is discarded.

- C's system() function
  
  If the target command or program of system() cannot be started, the system load service return code is returned as the function value of system(). Otherwise, the return code of the created enclave is reported as the function value of system(), and the reason code is discarded.

- 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

If the child enclave opens a message file, the file is closed when that enclave terminates.

OpenExtensions 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).

- The spawn() function is only allowed from a POSIX(ON) enclave. This applies to implicit spawn() resulting from a system() mapped to a spawn(), and to explicit spawn() functions.

- C exec() can be issued only from a single-thread enclave.

Any violations of the above restrictions result in a severity 3 condition being generated.

AMODE Considerations

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).
Part 2. Language Environment Debugging Guide
Chapter 9. Debugging C/C++ Routines

The information that follows is aditional for use with z/VM when using z/OS.

Language Environment Debugging Guide

This chapter provides specific information to help you debug applications that contain one or more C/C++ routines.

Debugging C/C++ Input/Output Programs

You can use C/C++ conventions such as __amrc and perror() when you debug I/O operations.

__last_op Values

The __last_op field is the most important of the __amrc fields. It defines the last I/O operation C/C++ was performing at the time of the I/O error. You should note that the structure is neither cleared nor set by non-I/O operations, so querying this field outside of a SIGIOERR handler should only be done immediately after I/O operations. Table 29 lists __last_op values you could receive and where to look for further information.

Table 29. __last_op Values and Diagnosis Information

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__IO_INIT</td>
<td>Will never be seen by SIGIOERR exit value given at initialization.</td>
</tr>
<tr>
<td>BSAM_OPEN</td>
<td>Sets __error with return code from OS OPEN macro.</td>
</tr>
<tr>
<td>BSAM_CLOSE</td>
<td>Sets __error with return code from OS CLOSE macro.</td>
</tr>
<tr>
<td>BSAM_READ</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 66) filled in).</td>
</tr>
<tr>
<td>BSAM_NOTE</td>
<td>NOTE returned 0 unexpectedly, no return code.</td>
</tr>
<tr>
<td>BSAM_POINT</td>
<td>This will not appear as an error lastop.</td>
</tr>
<tr>
<td>BSAM_WRITE</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 65) filled in).</td>
</tr>
<tr>
<td>BSAM_CLOSE_T</td>
<td>Sets __error with return code from OS CLOSE TYPE=T.</td>
</tr>
<tr>
<td>BSAM_BDL</td>
<td>Sets __error with return code from OS BDL macro.</td>
</tr>
<tr>
<td>BSAM_STOW</td>
<td>Sets __error with return code from OS STOW macro.</td>
</tr>
<tr>
<td>TGET_READ</td>
<td>Sets __error with return code from TSO TGET macro.</td>
</tr>
<tr>
<td>TPUT_WRITE</td>
<td>Sets __error with return code from TSO TPUT macro.</td>
</tr>
<tr>
<td>IO_DEVTYPE</td>
<td>Sets __error with return code from I/O DEVTYPE macro.</td>
</tr>
<tr>
<td>IO_RDJFCB</td>
<td>Sets __error with return code from I/O RDJFCB macro.</td>
</tr>
<tr>
<td>IO_TRKCALC</td>
<td>Sets __error with return code from I/O TRKCALC macro.</td>
</tr>
<tr>
<td>IO_OBTAIN</td>
<td>Sets __error with return code from I/O CAMLST OBTAIN.</td>
</tr>
<tr>
<td>IO_LOCATE</td>
<td>Sets __error with return code from I/O CAMLST LOCATE.</td>
</tr>
<tr>
<td>IO_CATALOG</td>
<td>Sets __error with return code from I/O CAMLST CAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>IO_UNCATALOG</td>
<td>Sets __error with return code from I/O CAMLST UNCAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>IO_RENAME</td>
<td>Sets __error with return code from I/O CAMLST RENAME.</td>
</tr>
<tr>
<td>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__SVC99_ALLOC</td>
<td>Sets __alloc structure with info and error codes from SVC 99 allocation.</td>
</tr>
<tr>
<td>__SVC99_ALLOC_NEW</td>
<td>Sets __alloc structure with info and error codes from SVC 99 allocation of NEW file.</td>
</tr>
<tr>
<td>__SVC99_UNALLOC</td>
<td>Sets __unalloc structure with info and error codes from SVC 99 unallocation.</td>
</tr>
<tr>
<td>__C_TRUNCATE</td>
<td>Set when C or C++ truncates output data. Usually this is data written to a text file with no newline such that the record fills up to capacity and subsequent characters cannot be written. For a record I/O file this refers to an fwrite() writing more data than the record can hold. Truncation is always rightmost data. There is no return code.</td>
</tr>
<tr>
<td>__C_FCBCHECK</td>
<td>Set when C or C++ FCB is corrupted. This is due to a pointer corruption somewhere. File cannot be used after this.</td>
</tr>
<tr>
<td>__C_DBCS_TRUNCATE</td>
<td>This occurs when writing DBCS data to a text file and there is no room left in a physical record for anymore double byte characters. A new-line is not acceptable at this point. Truncation will continue to occur until an SI is written or the file position is moved. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_SO_TRUNCATE</td>
<td>This occurs when there is not enough room in a record to start any DBCS string or else when a redundant SO is written to the file before an SI. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_SI_TRUNCATE</td>
<td>This occurs only when there was not enough room to start a DBCS string and data was written anyways, with an SI to end it. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_UNEVEN</td>
<td>This occurs when an SI is written before the last double byte character is completed, thereby forcing C or C++ to fill in the last byte of the DBCS string with a padding byte X'FE'. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_CANNOT_EXTEND</td>
<td>This occurs when an attempt is made to extend a file that allows writing, but cannot be extended. Typically this is a member of a partitioned data set being opened for update.</td>
</tr>
<tr>
<td>__VSAM_OPEN_FAIL</td>
<td>Set when a low level VSAM OPEN fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_RRDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_KSDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS_PATH</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_KSDS_PATH</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_MODCB</td>
<td>Set when a low level VSAM MODCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_TESTCB</td>
<td>Set when a low level VSAM TESTCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
</tbody>
</table>
Table 29. __last_op Values and Diagnosis Information (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__VSAM_SHOWCB</td>
<td>Set when a low level VSAM SHOWCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_GENCB</td>
<td>Set when a low level VSAM GENCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_GET</td>
<td>Set when the last op was a low level VSAM GET; if the GET fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_PUT</td>
<td>Set when the last op was a low level VSAM PUT; if the PUT fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_POINT</td>
<td>Set when the last op was a low level VSAM POINT; if the POINT fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_ERASE</td>
<td>Set when the last op was a low level VSAM ERASE; if the ERASE fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_ENDREQ</td>
<td>Set when the last op was a low level VSAM ENDREQ; if the ENDREQ fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_CLOSE</td>
<td>Set when the last op was a low level VSAM CLOSE; if the CLOSE fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__QSAM_GET</td>
<td>__error is not set (if abend (errno == 92), __abend is set, otherwise if read error (errno == 66), look at __msg.</td>
</tr>
<tr>
<td>__QSAM_PUT</td>
<td>__error is not set (if abend (errno == 92), __abend is set, otherwise if write error (errno == 65), look at __msg.</td>
</tr>
<tr>
<td>__QSAM_TRUNC</td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td>__QSAM_FREEPOOL</td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td>__QSAM_CLOSE</td>
<td>Sets __error to result of OS CLOSE macro.</td>
</tr>
<tr>
<td>__QSAM_OPEN</td>
<td>Sets __error to result of OS OPEN macro.</td>
</tr>
<tr>
<td>__CMS_OPEN</td>
<td>Sets __error to result of FSOPEN.</td>
</tr>
<tr>
<td>__CMS_CLOSE</td>
<td>Sets __error to result of FSCLOSE.</td>
</tr>
<tr>
<td>__CMS_READ</td>
<td>Sets __error to result of FSREAD.</td>
</tr>
<tr>
<td>__CMS_WRITE</td>
<td>Sets __error to result of FSWRITE.</td>
</tr>
<tr>
<td>__CMS_STATE</td>
<td>Sets __error to result of FSSTATE.</td>
</tr>
<tr>
<td>__CMS_ERASE</td>
<td>Sets __error to result of FSERASE.</td>
</tr>
<tr>
<td>__CMS_RENAME</td>
<td>Sets __error to result of CMS RENAME command.</td>
</tr>
<tr>
<td>__CMS_EXTRACT</td>
<td>Sets __error to result of DMS EXTRACT call.</td>
</tr>
<tr>
<td>__CMS_LINERD</td>
<td>Sets __error to result of LINERD macro.</td>
</tr>
<tr>
<td>__CMS_LINEWRT</td>
<td>Sets __error to result of LINEWRT macro.</td>
</tr>
<tr>
<td>__CMS_QUERY</td>
<td>__error is not set.</td>
</tr>
<tr>
<td>__HSP_CREATE</td>
<td>Indicates last op was a DSPSERV CREATE to create a hiperspace for a hiperspace memory file. If CREATE fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__HSP_DELETE</td>
<td>Indicates last op was a DSPSERV DELETE to delete a hiperspace for a hiperspace memory file during termination. If DELETE fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__HSP_READ</td>
<td>Indicates last op was a HSPSERV READ from a hiperspace. If READ fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__HSP_WRITE</td>
<td>Indicates last op was a HSPSERV WRITE to a hiperspace. If WRITE fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__HSP_EXTEND</td>
<td>Indicates last op was a HSPSERV EXTEND during a write to a hiperspace. If EXTEND fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__CICS_WRITEQ_TD</td>
<td>Sets __error with error code from EXEC CICS WRITEQ TD.</td>
</tr>
<tr>
<td>__LFS_OPEN</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_CLOSE</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_READ</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_WRITE</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_LSEEK</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_FSTAT</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
</tbody>
</table>

### Using __errno2() to Diagnose Application Problems

Use __errno2() when diagnosing problems in a z/OS UNIX or z/VM OpenExtensions application. This function enables C/C++ application programs to access diagnostic information returned to the C/C++ run-time library from an underlying kernel callable service. __errno2() returns the reason code of the last failing kernel callable service called by the C/C++ run-time library. The returned value is intended for diagnostic display purposes only. The function call is always successful.

**Note:** Since the __errno2() function returns the reason code of the kernel callable service that last failed, and not all function calls invoke the kernel, the value returned by __errno2() may be misleading.
Figure 7 is an example of a routine using __errno2().

```c
#include <stdio.h>
#include <errno.h>
FILE *myfopen(const char *fn, const char *mode) {
    FILE *f;
    f = fopen(fn, mode);
    if (f == NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    }
    return(f);
}
```

Figure 7. Example of a Routine Using __errno2()

Figure 8 is an example of a routine using the environment variable _EDC_ADD_ERRNO2, and Figure 9 shows the sample output from that routine.

```c
#include <stdio.h>
#include <errno.h>
int main(void) {
    FILE *fp;
    /* add errno2 to perror message */
    setenv("_EDC_ADD_ERRNO2","1",1);
    fp = fopen("testfile.dat", "r");
    if (fp == NULL)
        perror("fopen error");
}
```

Figure 8. Example of a Routine Using _EDC_ADD_ERRNO2

fopen error: EDC5129I No such file or directory.
(errno2=0x05620062)

Figure 9. Sample Output of a Routine Using _EDC_ADD_ERRNO2

### Generating a Language Environment Dump of a C/C++ Routine

You can use either the CEE3DMP callable service or the cdump(), csnap(), and ctrace() C/C++ functions to generate a Language Environment dump of C/C++ routines. These C/C++ functions call CEE3DMP with specific options.

#### cdump()

You can generate useful diagnostic information by using the cdump() function. cdump() produces a main storage dump with the activation stack. This is equivalent to calling CEE3DMP with the option string: TRACEBACK BLOCKS VARIABLES FILES STORAGE STACKFRAME(ALL) CONDITION ENTRY.

When cdump() is invoked from a user routine, the C/C++ library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of cdump() results in a SNAP identifier of 0. For each successive invocation, the ID is increased by one to a maximum of 256, after which the ID is reset to 0.

Under z/VM, the definition statement is:

```c
FILEDEF CEESNAP PRINTER (NOCHANGE PER
If the data set is not defined, or is not usable for any reason, `cdump()` returns a failure code of 1. This occurs even if the call to CEE3DMP is successful.

If the SNAP is not successful, the CEE3DMP DUMP file displays the following message:

Snap was unsuccessful

If the SNAP is successful, CEE3DMP displays this message:

Snap was successful; snap ID = nnn

Where `nnn` corresponds to the SNAP identifier described above. An unsuccessful SNAP does not result in an incrementation of the identifier.

Because `cdump()` returns a code of 0 only if the SNAP was successful or 1 if it was unsuccessful, you cannot distinguish whether a failure of `cdump()` occurred in the call to CEE3DMP or SNAP. A return code of 0 is issued only if both SNAP and CEE3DMP are successful.

A successful SNAP results in a large quantity of output. In addition to a SNAP dump, an Language Environment formatted dump is also taken.

csnap()

The `csnap()` function produces a condensed storage dump. `csnap()` is equivalent to calling CEE3DMP with the option string: TRACEBACK FILES BLOCKS VARIABLES NOSTORAGE STACKFRAME(ALL) CONDITION ENTRY.

To use these functions, you must add `#include <ctest.h>` to your C/C++ code. The dump is directed to output `dumpname`, which is specified in a FILEDEF CEEDUMP command in z/VM.

`cdump()`, `csnap()`, and `ctrace()` all return a 1 code in the SPC environment because they are not supported in SPC.

For more details about the syntax of these functions, refer to the [XL C/C++ for z/VM: Runtime Library Reference](https://www.ibm.com).
Chapter 10. Diagnosing Problems with Language Environment

The information that follows is additional for use with z/VM when using [Language Environment Debugging Guide](LanguageEnvironmentDebuggingGuide).

This chapter provides information for diagnosing problems in the Language Environment product. It helps you determine if a correction for a product failure similar to yours has been previously documented. If the problem has not been previously reported, it tells you how to open a Problem Management Record (PMR) to report the problem to IBM, and if the problem is with an IBM product, what documentation you need for an Authorized Program Analysis Report (APAR).

**Diagnosis Checklist**

Step through each of the items in the diagnosis checklist below to see if they apply to your problem. The checklist is designed to either solve your problem or help you gather the diagnostic information required for determining the source of the error. It can also help you confirm that the suspected failure is not a user error; that is, it was not caused by incorrect usage of the Language Environment product or by an error in the logic of the routine.

1. If your failing application contains programs that were changed since they last ran successfully, review the output of the compile or assembly (listings) for any unresolved errors.
2. If there have not been any changes in your applications, check the output (console logs) for any messages from the failing run.
3. Check the message prefix to identify the component that issued the message. This can help you determine the cause of the problem. Following are some of the prefixes and their respective origins.
   - **EDC** The prefix for C/C++ messages. The following series of messages are from the C/C++ run-time component of Language Environment: 5000 (except for 5500, which are from the DSECT utility), 6000, and 7000.
   - **IGZ** The prefix for messages from the COBOL run-time component of Language Environment.
   - **IBM** The prefix for messages from the PL/I run-time component of Language Environment.
   - **CEE** The prefix for messages from the common run-time component of Language Environment.
4. For any messages received, check for recommendations in the “Programmer Response” sections of the messages in this manual.
5. Verify that abends are caused by product failures and not by program errors. See the appropriate chapters in this manual for a list of Language Environment-related abend codes.
6. Your installation may have received an IBM Program Temporary Fix (PTF) for the problem. Verify that you have received all issued PTFs and have installed them, so that your installation is at the most current maintenance level.
7. The preventive service planning (PSP) bucket, an online database available to IBM customers through IBM service channels, gives information about product installation problems and other problems. Check to see whether it contains information related to your problem.
8. Narrow the source of the error.
• If a Language Environment dump is available, locate the traceback in the Language Environment dump for the source of the problem.
• If a system dump is taken on z/VM, follow the save area chain to find out the name of the failing module and whether IBM owns it. For information on finding the routine name, see z/OS: Language Environment Debugging Guide.

9. After you identify the failure, consider writing a small test case that re-creates the problem. The test case could help you determine whether the error is in a user routine or in the Language Environment product. Do not make the test case larger than 75 lines of code. The test case is not required, but it could expedite the process of finding the problem.

If the error is not a Language Environment failure, refer to the diagnosis procedures for the product that failed.

10. Record the conditions and options in effect at the time the problem occurred. Compile your program with the appropriate options to obtain an assembler listing and data map. If possible, obtain the LOAD/GENMOD map if running on z/VM. Note any changes from the previous successful compilation or run. For an explanation of compiler options, refer to the compiler-specific programming guide.

11. If you are experiencing a no-response problem, try to force a dump. Under z/VM in the CP mode, enter the DUMP command.

12. Record the sequence of events that led to the error condition and any related programs or files. It is also helpful to record the service level of the compiler associated with the failing program.
Part 3. Language Environment Run-Time Messages
Chapter 11. C/C++ Run-Time Messages

The information that follows is additional for use with z/VM when using z/OS. Language Environment Run-Time Messages.

The following run-time messages pertain to C/C++. Each message is followed by an explanation describing the condition that caused the message, a programmer response suggesting how you might prevent the message from occurring again, and a system action indicating how the system responds to the condition that caused the message.

The messages also contain a symbolic feedback code, which represents the first 8 bytes of a 12-byte condition token. You can think of the symbolic feedback code as the nickname for a condition. As such, the symbolic feedback code can be used in user-written condition handlers to screen for a given condition, even if it occurs at different locations in an application.

The messages in this section contain alphabetic suffixes that have the following meaning:
- I Informational message
- W Warning message
- E Error message
- S Severe error message
- C Critical error message

**EDC5230I** ESM error.

**Explanation:** An internal External Security Manager (ESM) error occurred. This message is equivalent to the OS/390 UNIX System Services errno ECMSESMERR.

**Programmer response:** Report this problem to your system programmer.

**System action:** Messages are displayed on the file pool server operator console indicating the error and z/VM processing continues.

**Problem determination:** EDC53E

---

**EDC6000E** The raise() function was issued for the signal SIGFPE.

**Explanation:** The program has invoked the raise() function with the SIGFPE signal specified and the default action specified.

**Programmer response:** None.

**System action:** The program is terminated and a traceback or dump is issued, depending on the TERMTHDACT run-time option. A return code of 3000000 is returned.

**Problem determination:** EDC5RI

---

**EDC6002E** The raise() function was issued for the signal SIGSEGV.

**Explanation:** The program has invoked the raise() function with the SIGSEGV signal specified and the default action specified.

**Programmer response:** None.

**System action:** The program will be terminated and a traceback or dump is issued, depending on the TERMTHDACT run-time option. A return code of 3000000 is returned.

**Problem determination:** EDC5RI

---

**EDC6003E** The raise() function was issued for the signal SIGABND.

**Explanation:** The program has invoked the raise() function with the SIGABND signal specified and the default action specified.
Programmer response: None.
System action: The program will be terminated and a traceback or dump is issued, depending on the TERMTHDACT run-time option. A return code of 3000000 is returned.
Problem determination: EDC5RN

EDC6004E The raise() function was issued for the signal SIGTERM.
Explanation: The program has invoked the raise() function with the SIGTERM signal specified and the default action specified.
Programmer response: None.
System action: The program will be terminated and a traceback or dump is issued, depending on the TERMTHDACT run-time option. A return code of 3000000 is returned.
Problem determination: EDC5RJ

EDC6005E The raise() function was issued for the signal SIGINT.
Explanation: The program has invoked the raise() function with the SIGINT signal specified and the default action specified.
Programmer response: None.
System action: The program will be terminated and a traceback or dump is issued, depending on the TERMTHDACT run-time option. A return code of 3000000 is returned.
Problem determination: EDC5RK

EDC6006E The raise() function was issued for the signal SIGABRT.
Explanation: The program has invoked the raise() function with the SIGABRT signal specified and the default action specified.
Programmer response: None.
System action: The program will be terminated and a traceback or dump is issued, depending on the TERMTHDACT run-time option. A return code of 2000000 is returned.
Problem determination: EDC5RM

EDC6007E The raise() function was issued for the signal SIGUSR1.
Explanation: The program has invoked the raise() function with the SIGUSR1 signal specified and the default action specified.
Programmer response: None.
System action: The program will be terminated and a traceback or dump is issued, depending on the TERMTHDACT run-time option. A return code of 3000000 is returned.
Problem determination: EDC5RP

EDC6008E The raise() function was issued for the signal SIGUSR2.
Explanation: The program has invoked the raise() function with the SIGUSR2 signal specified and the default action specified.
Programmer response: None.
System action: The program will be terminated and a traceback or dump is issued, depending on the TERMTHDACT run-time option. A return code of 3000000 is returned.
Problem determination: EDC5RO

EDC6009E The raise() function was issued for the signal SIGIOERR.
Explanation: The program has invoked the raise() function with the SIGIOERR signal specified and the default action specified.
Programmer response: None.
System action: The program will be terminated and a traceback or dump is issued, depending on the TERMTHDACT run-time option. A return code of 3000000 is returned.
Problem determination: EDC5RP

84 z/VM: Language Environment User's Guide
Chapter 12. COBOL Run-Time Messages

The information that follows is additional for use with z/VM when using Language Environment Run-Time Messages.

The following messages pertain to COBOL. Each message is followed by an explanation describing the condition that caused the message, a programmer response suggesting how you might prevent the message from occurring again, and a system action indicating how the system responds to the condition that caused the message.

The messages also contain a symbolic feedback code, which represents the first 8 bytes of a 12-byte condition token. You can think of the symbolic feedback code as the nickname for a condition. As such, the symbolic feedback code can be used in user-written condition handlers to screen for a given condition, even if it occurs at different locations in an application.

The messages in this section contain alphabetic suffixes that have the following meaning:

- **I** Informational message
- **W** Warning message
- **E** Error message
- **S** Severe error message
- **C** Critical error message

For more COBOL run-time messages, see z/OS: Language Environment Run-Time Messages

<table>
<thead>
<tr>
<th>IGZ0189S</th>
<th>Program pgmname cannot be run in this operating system environment.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Explanation:</strong></td>
<td>The program contains features that are not supported in this operating system environment. For example, when running on CMS, the following features are not supported:</td>
</tr>
<tr>
<td></td>
<td>- programs compiled with the DLL compiler option</td>
</tr>
<tr>
<td></td>
<td>- programs compiled with the ARITH(EXTEND) compiler option</td>
</tr>
<tr>
<td></td>
<td>- programs compiled with Enterprise COBOL for z/OS and OS/390 V3R1 and later</td>
</tr>
<tr>
<td><strong>Programmer response:</strong></td>
<td>Modify the program to use supported features for the environment or run the program in the appropriate environment.</td>
</tr>
<tr>
<td><strong>System action:</strong></td>
<td>The application was terminated.</td>
</tr>
<tr>
<td><strong>Problem determination:</strong></td>
<td>IGZ05T</td>
</tr>
</tbody>
</table>
Part 4. Customizing
Chapter 13. Customizing

Once the product has been installed, it can be customized using the CUSTLE EXEC. This EXEC will do the following:

1. Prompt you for the area you wish to customize:
   - Runtime Options
   - User Exit Options
   - 'C' Component Locale Time Information
   - Saved Segments Components
   - COBOL Reusable environment
2. Invoke an 'XEDIT' session for the specific customization component requested;
3. Re-assemble, if required, component customized;
4. Rebuild required modules using the specific VMSES/E part handler.

**Note:** The CUSTLE EXEC requires that the High Level Assembler program (HLASM) be available. It must be on a disk that you have accessed as A, B, C, D, S or Y.

To run the CUSTLE EXEC you must be on a userid that has access to the VMSES code (the default disk is the MAINT 5E5 disk). If you are logged on to the MAINT userid, the 5E5 is normally accessed as filemode B. Specify a PPF name (such as SERVP2P or ZVM) and the LE component name (usually LE or LESFS). The screen shown in Figure 10 is displayed:

```
Language Environment for z/VM
Version 6 Release 2 Mod 0

1) Run Time Options
2) User Exits
3) 'C' Locale Time Info
4) Named Saved Segments (NSS)
5) COBOL Reusable Environment

Enter number of option you wish to change or
Enter "END or QUIT" to Exit the customization.
```

*Figure 10. Customization EXEC - Panel 1*

The screen offers menu choices for run time options, user exits, and other information.

**Updating Run-Time Options**

Run-time options are updated by invoking the customization EXEC which puts you into an XEDIT session of CEEDOPT ASSEMBLE. After you update and file CEEDOPT, the EXEC assembles it (using HLASM) and if the assembly is successful, will then rebuild the modules in which it is included. Modules which will be rebuilt are CEBINIT, CEEPICI, CEEPIPI, and CEEEPLPKA, all of which are in Build List "CEEBLMOD". See **z/OS: Language Environment Programming** Reference for more information. Once the modules have been rebuilt, you will be reminded to rebuild the SCEE/SCEEEX segments.
Updating User Exit Options

The assembler user exit is updated by invoking the customization EXEC which puts you into an XEDIT session of CEEBXITB ASSEMBLE. After you update and file CEEBXITB, the EXEC assembles it (using HLASM) and if the assembly is successful, will then rebuild the component in which it is included. Modules which will be rebuilt are CEEBINIT, CEEBPICI and CEEPIPI all of which are in Build List "CEEBLMOD". Once the modules have been rebuilt, you will be reminded to rebuild the SCEE/SCEEX segments.

C Component Locale Time Information

Note

Due to the size and having to rebuild the SCEERUN LOADLIB for this option, your "A" disk, which z/VM uses as an interim work disk during the rebuild, must have at least 20 cylinders on a 3390, or equivalent, of unused (free) space.

C locale time information is used for options such as Time Zone name and Daylight Savings Time starting dates.

Locale time is updated by editing a file named 'EDCLOCI'. The EXEC will put you into an XEDIT session of EDCLOCI ASSEMBLE and after updates are completed it is filed and then assembled using HLASM. Once successfully assembled, the EXEC will rebuild the required components and the C locale time is updated. Once the modules have been rebuilt, you will be reminded to rebuild the SCEE/SCEEX segments.

Updating Saved Segments

After successfully installing Language Environment, you can load certain routines into Saved Segments on z/VM. Placing routines into Saved Segments reduces overall system storage requirements by making the routines sharable. Also, initiation/termination (init/term) time is reduced for each application, since load time decreases.

LE includes two build lists, CEEBLSGA and CEEBLSGB, plus the necessary LSEG files required to install specific routines of LE into segments. By selecting option 4 in the customization exec, these individual build lists can be tailored to load only specific routines of the LE component (for example, commonly used COBOL, PL/I, or C routines) into segments. Each build list contains comments that identify these routines and help tailor the segment install.

Customizing can be accomplished by either commenting or uncommenting the appropriate LOADFUNC component statement(s) or by adding new LOADFUNC statements into the build list. An asterisk (*) inserted in the first column of any LOADFUNC statement will eliminate that component from being included while deleting one from the first column will include the component. In the following example, the PL/I routines (IBMRLIB1, IBMRCOMP, and IBMRPTLA) which are normally installed below the line, and thus included in the CEEBLSGB build list, will be eliminated from the saved segment environment.

*********************************************************************
* LANGUAGE ENVIRONMENT for z/VM *
* Version 6 Release 2 Modification 0 *
*
To reinstate routines in the saved segments, remove the asterisk and regenerate the segments. To include other routines in saved segments, add the appropriate LOADFUNC statement into the respective build list.

Updating the COBOL Component Reusable Environment

COBOL's reusable environment behavior is updated by invoking the customization EXEC which puts you into an XEDIT session of IGZERREO ASSEMBLE. After you update and file IGZERREO, the EXEC assembles it (using HASM) and if the assembly is successful, will then prompt you to see if you want to rebuild the component in which it is included. The module that will be rebuilt is CEEEV005 which is in build list IGZBLMOD.

The COBOL reusable environment behavior can be modified to control how program checks are handled when they occur in a non-Language Environment-conforming driver. The COBOL reusable environment is established with the RTEREUS run-time option or a call to either ILBOSTP0 or IGZERRE INIT.

With the IBM-supplied default setting for COBOL's reusable environment behavior (IGZERREO with REUSENV=COMPAT), when a program check occurs while the reusable environment is dormant (that is, between a GOBACK from a top level COBOL program to the non-Language Environment conforming assembler driver and the next call to a COBOL program), a S0Cx abend will occur. This behavior is compatible with the VS COBOL II and OS/VS COBOL run-times, but it significantly impacts the performance when a COBOL/370 or COBOL for MVS & VM program is invoked repeatedly in a COBOL reusable environment. The performance degradation is caused by Language Environment issuing an ESPIE RESET when the reusable environment becomes dormant and then an ESPIE SET upon reentering the reusable environment.
COBOL’s reusable environment behavior can be modified (IGZERREO with REUSENV=OPT) so that all program checks will be intercepted by Language Environment, even those that occur while the reusable environment is dormant. In this case, a program check that occurs while the reusable environment is dormant will result in a 4036 abend from Language Environment. However, since Language Environment does not have to issue the ESPIE RESET and ESPIE SET between invocations of the COBOL program, this can be faster than using REUSENV=COMPAT.

Modifying the behavior of the COBOL Reusable Environment

Modify the IGZREOP macro invocation, depending on the function that you want. To run with VS COBOL II and OS/VS COBOL run-time compatibility mode (that is, the user has control of program checks that occur when the COBOL reusable environment is dormant, resulting in an additional performance cost), use:

IGZREOP REUSENV=COMPAT

To run with optimum performance (Language Environment intercepts all program checks that occur when the COBOL reusable environment is dormant and converts them to a 4036 abend, resulting in improved performance), use:

IGZREOP REUSENV=OPT
Part 5. Appendixes
Appendix A. Prelinking an Application

This appendix describes how to prelink your programs under Language Environment. Unless otherwise indicated, the prelinking process applies to C and COBOL in z/VM.

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 available to be able to directly receive the output of the C, COBOL, and PL/I compilers, thus eliminating the requirement for the prelink step. The advantage of using the binder is that the resulting executable program is fully rebindable. For information on how to use the binder, see z/VM: Program Management Binder for CMS.

For information on how to build and use DLLs, see z/OS: Language Environment Programming Guide.

Which Programs Need to Be Prelinked

The prelink step is required when an executable program is built on z/VM, or if it utilizes the system programming facilities of C. The prelink step may be eliminated since the binder can handle the output of the C and COBOL 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 C compiler creates GOFF object code when the XPLINK compiler option is specified.

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
- Programs which utilize the system programming facilities of C.
- Non-XPLINK C programs compiled with any of the following compiler options:
  - RENT
  - LONGNAME
  - DLL
- COBOL programs compiled with any of the following compiler options:
  - PGMNAME(LONGMIXED)
  - PGMNAME(LONGUPPER)
- C programs compiled to run under OpenExtensions for z/VM

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 z/OS: Language Environment Programming Guide.

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 run-time initialization, including data initialization for C and DLL initialization information.
- For C object modules compiled with RENT, 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 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 object modules specified on the command line
- Secondary input: input from automatic library calls
- Input specified in one or more INCLUDE control statements processed in primary and secondary input

The process of resolving or including input from these sources depends on the type of the source, and the current input and prelink options.

Primary Input

When an object module name SNAME is specified on the command line, the following process occurs:
- If it exists, it is immediately resolved by reading SNAME TEXT.
- If the LIBE option is in effect, then SNAME is immediately resolved by reading the member of the first TXTLIB found in the GLOBAL list that has the same member name, or alias name.
- If SNAME is still unresolved, it could be subsequently resolved if a defined function or variable called SNAME is encountered in input.

INCLUDE Control Statements

For the INCLUDE ddname() and INCLUDE ddname(member) forms, an attempt is made to read the ddname or member of the ddname (whichever is specified). This request is resolved if the read is successful.
For the INCLUDE SNAME form, the input is resolved using the same algorithm as for primary input.

See "INCLUDE Control Statement" on page 102 for a description of the INCLUDE control statement.

References to Currently Undefined Symbols (External References)

If, during the automatic library call, the symbol was not found to be the name of an existing TXTLIB library routine or TEXT file, then the symbol can subsequently be defined if a function or variable with the same name is encountered.

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 103 for a complete description of the RENAME control statement.

C only: If the symbol is an L-name that was not resolved by previous automatic library call and also corresponds to a C library function or object, the symbol is resolved under the S-name of the symbol. For example, if you do not supply a version of printf(), an attempt would be made to find and use PRINTF in its place as the C library only ships PRINTF.

Unresolved requests generate error or warning messages to the terminal or to the prelinker map.

Writable static references which 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 cannot be used as input to another prelink.

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 options.

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 AUTO command option is in effect and the reference is not to static external data, SNAME TEXT is read.
  − If the LIBE command option is in effect, the GLOBAL TXTLIBs are searched in order as follows:
    1. If the TXTLIB contains a C370LIB-directory created using the Object Library Utility, and the C370LIB-directory indicates that a defined symbol by that name exists, the member of the TXTLIB containing that symbol is read.
    2. If the TXTLIB 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 LIBE command option is in effect, the GLOBAL TXTLIBs are searched.
    If the TXTLIB contains a C370LIB-directory created using the Object Library

Appendix A. Prelinking an Application 97
Utility, and the C370LIB-directory indicates that a defined symbol by that
name exists, the member of the TXTLIB indicated as containing that symbol is
read.

**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 Figure 11 on page
[99](#) shows, the prelinker map contains several individual sections that are only
generated if they are applicable.
### Prelincking Applications

**Prelinker Map**

```
PLINK:5739A03 V4 R4 M00 IBM z/VM 2003/05/20 13:45:16
```

Command Options.....: NONCAL NOMEMORY ER DUP MAP
: NOOMVS NOUPCASE

**Object Resolution Warnings**

WARNING EDC4015: Unresolved references are detected:
CEESTART @ORGLOR CEESGO03

**File Map**

```
*ORIGIN FILE ID FILE NAME
P 00001 DD:SYSIN
IN 00002 *** DESCRIPTORS ***
*ORIGIN: P=primary input PI=primary INCLUDE SI=secondary INCLUDE
A=automatic call R=RENAME card L=C Library
IN=internal
```

**Writable Static Map**

```
OFFSET LENGTH FILE ID INPUT NAME
0 4 00001 this_int_is_in_writable_static
8 10 00002 <year>
```

**Load Module Map**

```
MODULE ID MODULE NAME
00001 EXPONLY
```

**Import Symbol Map**

```
*TYPE FILE ID MODULE ID NAME
D 00001 00001 year
*TYPE: D=imported data C=imported code
```

Figure 11. Prelinker Map (Part 1 of 2)
The numbers in the following text correspond to the numbers shown in the map.

1 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.

2 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.

3 File Map
   This section lists the object modules that were included in input. An object module consisting only of RENAME control statements, for example, is not shown. Also provided in this section are source origin (*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.
Prelinking Applications

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.

4 Writable Static Map
This section is generated if an object module was encountered that contains defined static external data. This area also contains variable descriptors for any imported variables and, if required, function descriptors. This section lists the names of such objects, their lengths, their relative offset within the writable static area, and a FILE ID for the file containing the object's definition.

Imported variables and DLL-referenced functions have angular brackets (<>) around their names in this section.

5 Load Module Map
This section is generated if the application imports symbols from other load modules. This section lists the names of the load modules.

6 Import Symbol Map
This section 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.

7 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, or COBOL EXPORTALL compiler option or C/C++ code containing the #pragma export directive generates an object module that exports symbols.

8 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 CMS file.

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, which is supported only under OS/390, has the following syntax:

Syntax

```
IMPORT
```

```
CODE dll-name function
DATA dll-name variable
```

```
SM590000
```

```
SM630000
```

`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 an HFS 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:

Syntax

```
INCLUDE
```

```
filename,
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
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:

Syntax

```
LIBRARY name (member) (external)
```

name
The ddname defining a loadlib. The ddname can point to an archive file in the BFS if the OE option is specified, or a CMS loadlib.

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.

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:

Syntax

```
RENAME L-name S-name
```

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.
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") 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."
  - 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++) 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 run-time 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 103.

2. If the L-name was found to have a corresponding S-name, the same name is chosen. For example, DOTOTALS 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 SPRINTF 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 UPCASE 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 UPCASE 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.

---

**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

Use the CPLINK EXEC to start the prelinker. The syntax for the CPLINK EXEC is:
Prelinking Applications

Syntax

```
CPLINK textdeck

(textdeck
  The file name of an input file to the prelinker. You can specify more than one
  file as input to the CPLINK EXEC. Each input file must be a COBOL object
  module or a C object module with the file type TEXT (that is, a program
  compiled with the RENT option or a compiled program with no writable static). If
  you previously used the z/VM GLOBAL TXTLIB command, you can specify the
  name of a TXTLIB member as the file name. The first text deck must contain
  the function's main, or the fetchable routine.

(options
  An option or list of options to be passed to the prelinker. The prelinker is started
  with CMOD EXEC if you specify options for the prelinker by using the option
  CPLINK(options) or if you have specified the LONGNAME or RENT compiler
  option. See "Prelinker Options" on page 107 for a list of prelink options.

Output from the prelinker is placed in the file CPOBJ TEXT A. The prelinker map is
placed in the file CPOBJ RMAP A.

Examples

The following example prelinks the text decks ROUTER, SENDMSG and REPLYMSG and
places the output text deck in CPOBJ TEXT A. A writable static map is generated and
placed in CPOBJ RMAP A. Unresolved references are not processed.
CPLINK ROUTER SENDMSG REPLYMSG (NOLIBE

The following example prelinks the text decks SORT, MERGE and READFILE and
displays only warning and error messages at the terminal. A prelink listing is not
generated. All external references are resolved from ACCNT TXTLIB and a disk
search.
GLOBAL TXTLIB ACCNT
CPLINK SORT MERGE READFILE (NOMAP NOER NODUP LIBE AUTO

To use CMOD to invoke the prelinker with the prelink options AUTO and the CMOD
option AUTO specify:
CMOD PGM (CPLINK(AUTO) AUTO

The following example shows you how to quickly link and prelink using CMOD while
keeping the CPOBJ TEXT text deck generated by the prelinker, without generating
either a LOAD or a CPLINK MAP.
CMOD MYMAIN MYPROCS ( CPLINK(KEEP) NOMAP
```
The following table describes the Language Environment prelinker options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO</td>
<td>NOAUTO</td>
</tr>
<tr>
<td>DLLNAME(dll-name)</td>
<td>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 TEMPNNAME, and the prelinker issues a warning.</td>
</tr>
<tr>
<td>DUP</td>
<td>NODUP</td>
</tr>
<tr>
<td>ER</td>
<td>NOER</td>
</tr>
<tr>
<td>LIBE</td>
<td>NOLIBE</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>
<tr>
<td>OE</td>
<td>NOOE</td>
</tr>
<tr>
<td>UPCASE</td>
<td>NOUPCASE</td>
</tr>
</tbody>
</table>
Prelinking Applications
Appendix B. Parameter List Formats

This appendix 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 below.

There are additional considerations depending on whether the main routine is in the C, C++, COBOL, or PL/I language. For information about parameter passing in these languages, see:

- "C and C++ Parameter Passing Considerations"
- "COBOL Parameter Passing Considerations" on page 113
- "PL/I Main Procedure Parameter Passing Considerations" on page 113

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 prior to entry into the main routine. Depending on how the parameters are passed, register 1 points on entry to the entities illustrated in Figure 12:

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 12. Some Alternate C/C++ Parameter Passing Styles

The first arrangement in Figure 12 can be used only for parameters that are integers.

A C main routine elects to use one of the styles shown in Figure 12 by specifying the PLIST(OS) run-time option in #pragma runopts (see "C PLIST and EXECOPS Interactions" on page 111); a C++ routine elects to use one of the styles with the
Operating System and Subsystem Parameter Lists

PLIST(OS) compiler option. 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 12 on page 109.

__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 12 on page 109. You must include the header file stdlib.h when using __osplist.

The third style is also currently 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 13 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 13. Accessing Parameters Using Macros __R1 and __osplist

Suitable casting and dereferencing are required when using these macros, as shown in Figure 14 on page 111 according to the parameter passing style in use.
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 run-time options on the command line at application invocation. NOEXECOPS indicates that run-time options cannot be so specified. When the EXECOPS run-time option is specified under MVS, Language Environment alters the MVS parameter list format: Language Environment removes any run-time 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:
  - If invoked by OSRUN, Language Environment uses the string presented in an MVS-like format located by the pointer held in register 1.
  - If not invoked by OSRUN, Language Environment uses the CMS extended parameter list.

- **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 run-time options are available. Register 1 is not interrogated by Language Environment.

The PLIST(HOST) setting permits portability of source code between MVS and z/VM. PLIST(HOST) allows the object to execute under z/VM (using either the MVS-format argument list for OSRUN or the extended argument list), 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, CMS, IMS, and TSO suboptions of PLIST for compatibility, use of PLIST(HOST) is recommended. There are some exceptions to this guideline:

**Preinitialization**

In the previous C interface to preinitialization, it was necessary to specify PLIST(MVS) in order to flag preinitialized routines. PLIST(MVS) is therefore still supported for compatibility.
Operating System and Subsystem Parameter Lists

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 run-time 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 31:

Table 31. Interactions of C PLIST and EXECOPS (#pragma runopts)

<table>
<thead>
<tr>
<th>Method of Invocation</th>
<th>PLIST Suboption</th>
<th>EXECOPS (default)</th>
<th>argc/argv</th>
<th><strong>R1</strong>/osplist and PCBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LKED, OSRUN</td>
<td>HOST</td>
<td>Yes. &lt;run-time options&gt; honored</td>
<td>argc = number of tokenized user args. argv[0...argc-1] = tokenized user args</td>
<td></td>
</tr>
<tr>
<td>LKED, OSRUN</td>
<td>HOST</td>
<td>No. &lt;run-time options&gt; ignored</td>
<td>argc = number of tokenized user args in both run-time options and user args argv[0...argc-1] = tokenized args in both run-time options and user args</td>
<td></td>
</tr>
<tr>
<td>Assembler calls C module with pre-Language Environment preinitialization PLIST with run-time options specified in the PLIST</td>
<td>MVS</td>
<td>Yes. &lt;run-time options&gt; honored</td>
<td>argc/argv = &lt;argc,argv&gt; structure specified for pre-Language Environment preinitialization</td>
<td></td>
</tr>
<tr>
<td>Assembler calls C module with pre-Language Environment preinitialization PLIST with run-time options specified in the PLIST</td>
<td>MVS</td>
<td>No. &lt;run-time options&gt; ignored</td>
<td>argc/argv = &lt;argc,argv&gt; structure specified for pre-Language Environment preinitialization</td>
<td></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 may 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 run-time 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:

- If the COBOL main is invoked by OSRUN, run-time options are removed. An adjusted string (without run-time options) is passed to the application.
- If the COBOL main is not invoked by OSRUN, register 0 points to a CMS extended parameter list. Run-time options are removed and repackaged as a halfword-prefixed string.
- If the COBOL main is invoked from an assembler routine using standard assembler linkage conventions, then register 1 and the argument list are 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, CMS (or CMSTPL for compatibility), CICS, IMS, or TSO. The SYSTEM option allows a program compiled under one system to run under another.

The NOEXECOPS procedure option indicates that run-time 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 run-time options might be present in the operating system parameter list. If present, these run-time options are removed by run-time initialization before the PL/I main procedure gains control.

In order for run-time options to be passed in the operating system parameter list for SYSTEM(MVS) or SYSTEM(CMS), 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 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 z/OS: Language Environment Programming Guide 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.
### Operating System and Subsystem Parameter Lists

Table 32. Interactions of SYSTEM and NOEXECOPS under z/VM

<table>
<thead>
<tr>
<th>SYSTEM Setting</th>
<th>No Run-Time Options (NOEXECOPS)</th>
<th>Run-Time Options Can Be Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM(CMS)</td>
<td>If the main procedure parameter is a single varying character string, a CMS extended parameter list is assumed and repackaged so the main procedure receives a halfword-prefixed string, without looking for run-time options. Otherwise, the parameter list is passed without change.</td>
<td>If the main procedure parameter is a single varying character string, a CMS extended parameter list is assumed and repackaged so the main procedure receives a halfword-prefixed string. Any run-time 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(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 run-time options are removed from the string, and the (potentially) altered string is passed. Otherwise, the parameter list is passed without change.</td>
</tr>
</tbody>
</table>

**Note:** When Language Environment is directed to use the CMS extended parameter list, and Language Environment determines that R0 is not pointing to a CMS extended parameter list, Language Environment issues user ABEND 4093, reason code X'60' (96).
Appendix C. Object Library Utility

The Object Library Utility is used to update libraries of object modules. A library is a text library (TXTLIB) 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.

Notes:
1. The TXTLIB command also creates object libraries but it does not allow you to include external names greater than 8 characters long. The syntax for the Object Library Utility is similar to the TXTLIB command.
2. Because C generates private code if you do not include a #pragma csect(code) directive in your source or if you do not create a NAME control statement using the ALIAS compiler option, you should use the FILENAME option on either the TXTLIB or C370LIB commands.

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

Use the C370LIB EXEC to create an object library:

Syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN libname fn</td>
<td>Creates a TXTLIB on your A-disk. If a TXTLIB with the same name already exists, it is replaced.</td>
</tr>
<tr>
<td>ADD libname fn</td>
<td></td>
</tr>
<tr>
<td>DEL libname membername</td>
<td></td>
</tr>
<tr>
<td>MAP libname</td>
<td></td>
</tr>
<tr>
<td>DIR libname</td>
<td></td>
</tr>
</tbody>
</table>

**GEN**

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

**libname**

Specifies the file name of a file of type TXTLIB that is to be created or listed, or from which members are to be added or deleted, or for which a Object Library Utility directory is to be built.
Object Library Utility

\textit{fn}  Specifies the name of file of type TEXT that you want to add to a TXTLIB.

\textbf{FILENAME}
Indicates that all the file names specified (\textit{fn1}) should be used as the member names for their respective entries in the TXTLIB file.

\textbf{ADD}
Adds TEXT files as members to an existing TXTLIB on a read/write disk. No checking is done for duplicate names, entry points, or CSECTs.

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

\textit{membername}
Specifies the name of a TXTLIB member that you want to delete.

\textbf{MAP}
Lists the names (entry points) of TXTLIB members. MAP produces a file, \textit{libname MAP}, on your A-disk. For more information about the map, see \textit{XL C/C++ for z/VM: User's Guide}.

\textbf{DIR}
Builds the TXTLIB Object Library Utility directory. The Object Library Utility directory contains the names (entry points) of library members. The DIR function is only necessary if TEXT files were previously added or deleted from the TXTLIB without using C370LIB.

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

When a TEXT file is added to a library, its member name is selected according to the following hierarchy:
1. From the file name, if the FILENAME option is specified
2. From the \textit{NAME} control statement, if present, in the TEXT file
3. From the file name

The CMS TXTLIB command’s \textbf{GEN}, \textbf{ADD}, and \textbf{DEL} functions are used as part of the C370LIB \textbf{GEN}, \textbf{ADD}, and \textbf{DEL} functions. Thus, any TXTLIB restrictions apply also to C370LIB unless otherwise stated. Members must be deleted by their member name. Any attempt to delete using a name other than the member name results in a warning message.

In the following example, the C programs \texttt{SUB1.C} and \texttt{SUB2.C} are compiled for L-names. The function library \texttt{SUBLIB TXTLIB A} is created with \texttt{SUB1 TEXT} using the \textit{GEN} command of C370LIB; the Object Library Utility \texttt{SUB2 TEXT} is added to the library using the \textit{ADD} command.

\begin{verbatim}
CC SUB1 (LO
CC SUB2 (LO
C370LIB GEN SUBLIB SUB1
C370LIB ADD SUBLIB SUB2
\end{verbatim}

\textbf{The LINKLOAD EXEC}
The following IBM-supplied EXEC generates a fetchable member of a z/VM load library:
Object Library Utility

Syntax

\[ \text{LINKLOAD textdeck LIB libname option} \]

\textit{textdeck}

Specifies the name of the input text decks. The file type of the object modules must be TEXT, and the source programs must have contained a \texttt{#pragma linkage (name, FETCHABLE)} preprocessor directive. Note that you do not specify the file type or the file mode when using the LINKLOAD EXEC.

\textit{libname}

Specifies the name of the library where the load member is to be stored. The library name parameter must be specified, but if it is the first parameter, the keyword LIB is optional.

\textit{option}

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

\textbf{CPLINK}

Allows you to pass options to the prelinker. The format of the CPLINK options is \texttt{CPLINK (prelinker options)}. CPLINK is called if it is required by the text decks, or if a CPLINK option is given. For more information see Appendix A, "Prelinking an Application," on page 95.

\textbf{MBR}

Indicates that the next argument, \textit{memname}, is the name of the member within the load library that is to be generated. If you do not specify a member name, the name of the text deck containing the fetchable code is used.

\textbf{LKED}

Indicates that the options following it are to be passed to LKED. If you do not use this option, default options are used. The format of the LKED keyword is \texttt{LKED (link-edit options)}. For more information on the LKED command, see "Link-Editing with the LKED Command" on page 14.

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

\textbf{ADD}

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

\textbf{REPLACE}

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

\textbf{NEW}

Indicates that an existing load library of the same name containing only the named member should be created.
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 15.

Object Library Utility Map

<table>
<thead>
<tr>
<th>Object Library Utility Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>C370LIB:5739A03 V6 R2 M00 IBM z/VM 2011/02/28 21:19:26</td>
</tr>
</tbody>
</table>

Library Name: TS41949.A.OBJECT 2011/02/28 21:19:26

*----------------------------------------------------------------------*
* Member Name: ASMSTUFF (D) 2011/02/28 21:19:26 2 *
*----------------------------------------------------------------------*
(S) External Name: CSECT1
(S) External Name: ENTRY1

*----------------------------------------------------------------------*
* Member Name: CSTUFF (D) 2011/02/28 21:19:26 2 *
*----------------------------------------------------------------------*
(L) Function Name: foo
(WL) External Name: this_int_is_in_writable_static_and_its_name_will_wrap_because_it_is_too_long

*----------------------------------------------------------------------*
* Member Name: CXXSTUFF (D) 2011/02/28 21:19:26 2 *
*----------------------------------------------------------------------*

User Comment: This is a user comment in CXXSTUFF

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

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

Figure 15. Object Library Utility Map

1 Map Heading
The heading contains the product number, the compiler release number, the compiler version number, and the date and time the Object Library Utility step commenced. The name of the library immediately follows the heading. To the right of the name of the library is the start time of the last Object Library Utility step that updated the Object Library Utility directory.

2 Member Heading
The name of the object module member is immediately followed by the ID of the processor that produced the object module. The processor ID is based on the presence of an END record in the object module having the processor information in the appropriate format. If this information is not present, the Processor ID field is not listed.
Object Library Utility

The Timestamp field is presented in \textit{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:

\begin{itemize}
  \item[(P)] Indicates that the timestamp is extracted from the object module from the date form of \texttt{#pragma comment} or from the timestamp form of \texttt{#pragma comment}, whichever comes first.
  \item[(D)] Indicates that the timestamp is based on the time that the Object Library Utility DIR command was last issued.
  \item[(F)] Indicates that the timestamp is the date of the object module file at the time the ADD or GEN command was issued for the member. This is applicable to z/VM only.
  \item[(T)] Indicates that the timestamp is the time that the ADD command was issued for the member. This is applicable to MVS only.
\end{itemize}

\section*{User Comments}

The user form of comments generated by \texttt{#pragma comment} is displayed. These comments are extracted from the END record. It is possible to manually add such comments on multiple END records and have them displayed in the listing. For more information on the END record, see \texttt{z/OS XL C/C++ Language Reference}.

\section*{Symbol Information}

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

\begin{itemize}
  \item ['L'] Indicates that the name is an L-name.
  \item ['S'] Indicates that the name is an S-name.
  \item ['W'] Indicates that this is a writable static object. If no 'W' is present, then this is not a writable static object.
  \item ['WL'] Indicates that this is both an L-name and in writable static.
\end{itemize}
Object Library Utility
Appendix D. Using the Systems Programming Environment

Note: This section 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, run-time 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.

• 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 chapter discusses how to build these applications once you have compiled them with the C compiler. Note that you must compile these programs with the NOSTART option.

Building Freestanding Applications

Freestanding applications need to be linked with specific alternate initialization routines. This is accomplished differently depending on which operating system you compiled your application under.

To explicitly include an alternative initialization routine under z/VM, include the TEXT file for the alternate entry point first in the LOAD commands. To include the alternate initialization routines described in this chapter, you must include SCEESPC in the GLOBAL TXTLIB list. For example, the commands in Figure 16 can be used to specify EDCXSTRT as an alternate initialization routine.

LOAD EDCXSTRT main-function (RESET EDCXSTRT ...)
GENMOD module-name (FROM EDCXSTRT

Figure 16. Specifying Alternate Initialization at Link-Edit

Building Freestanding Applications

When building freestanding applications under z/VM, SCEESPC TXTLIB must be made available (by the GLOBAL command) when issuing LOAD or INCLUDE commands. In addition to making SCEESPC TXTLIB available, you must specify NOSTART compiler option when compiling the file that contains the main function. This TXTLIB is not required at execution time.
Systems Programming Environment

The routines to support this function (EDCXSTRT and EDCXSTRL) are CEESTART replacements in your module. Therefore, the appropriate EDCXSTR{n TEXT file must be explicitly included first in the module.

A simple freestanding routine that does not require the Language Environment common library is shown in Figure 17. An example that requires the use of the Language Environment prelinker is shown in Figure 19.

```c
#include <stdlib.h>

int main() {
    static int i[5]={0,1,2,3,4};
    exit(4320+i[1]);
}
```

Figure 17. Simple Freestanding z/VM Routine

The z/VM commands required to build and run this routine are shown in Figure 18.

```
GLOBAL LOADLIB SCEERUN
CC RET54321 (NOSTART
GLOBAL TXTLIB SCEESPC
LOAD EDCXSTRT RET54321 (RESET EDCXSTRT
GENMOD RET54321 (FROM EDCXSTRT
RET54321
```

Figure 18. Building a Freestanding z/VM Routine

Special Considerations for Reentrant Modules

A simple freestanding routine that does not require the Language Environment common library is shown in Figure 19. This routine uses the exit() library function which, like sprintf(), is available to freestanding routines without requiring the Language Environment common 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(4320+i[1]);
}
```

Figure 19. Simple Reentrant Freestanding z/VM Routine

The commands required to build this routine are shown in Figure 20. The bracketed numbers in the figure refer to the comments that follow.

```
CC RETS4321 (NOSTART RENT ...
GLOBAL TXTLIB SCEESPC SCEELKED CMSLIB
[C20-1]
CLINK EDCXSTRT RETS4321 EDRCINT EDCXEXIT (MAP
[C20-2]
GLOBAL TXTLIB
[C20-3]
LOAD CPOBJ (MAP RESET EDCXSTRT
[C20-4]
GENMOD RETS4321 (FROM EDCXSTRT
```

Figure 20. Building a Reentrant Freestanding VM Routine

Notes

[C20-1] The TXTLIB CMSLIB is needed because CPLINK is a C program. The TXTLIB SCEESPC and SCEELKED are used to resolve external references.
The alternate initialization routine (EDCXSTRT in this example) must be included explicitly in the module. This should be the first CSECT in the module.

The routine EDCRCINT must be explicitly included in the module because the RENT compiler option is used. No error is detected at load time if this routine is not explicitly included. At run time, abend 2106, reason code 7205, results if EDCRCINT is required but not included.

EDCXEXIT must be explicitly included if the exit() function is used in the application.

No TXTLIB is required for further processing or execution of this module because no C library functions are needed.

EDCXSTRT must be specified as the module entry point.

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. SCEESPC TXTLIB must be available at link-edit. If C library functions are required by the exit routines, the libraries SCEELKED must also be made available after SCEESPC. If the routines were compiled with OPT(2), the entry point must be explicitly identified using the RESET option on the LOAD command.

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 LIBE option of the LOAD command causes the proper object modules to be included from SCEESPC TXTLIB.

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 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 121.

There are no special considerations for building user applications. The LIBE option of the load command causes the proper object modules to be included from SCEESPC TXTLIB. The automatic call facility causes the right routines from the TXTLIB (using the LIBE option) to be included.

Note: You must compile servers with the NOSTART option.
## Summary

**Table 33. 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>Run-Time 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, or an EXEC or CLIST.</td>
<td>EDCXSTRT must be explicitly included at bind time.</td>
<td>None.</td>
<td>Run-time options are specified by <code>#pragma runopts</code> in the compilation unit for the <code>main()</code> function. The HEAP and STACK options are honored. STACK defaults to above the 16M line.</td>
</tr>
<tr>
<td>A mainline function that requires C library functions.</td>
<td>From the command line, or an EXEC or CLIST.</td>
<td>EDCXSTRL must be explicitly included at bind time.</td>
<td>C library functions.</td>
<td>Run-time 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 by the application.</td>
<td>Run-time options are specified by <code>#pragma runopts</code> in the compilation unit for 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>Run-time options are specified by <code>#pragma runopts</code> in the compilation 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>
</tbody>
</table>
Table 33. 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>Run-Time Options and Other Considerations</th>
</tr>
</thead>
<tbody>
<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>Run-time 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>Run-time 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>The server and C library functions, if required by the server.</td>
<td></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>
Notices

This information was developed for products and services offered in the U.S.A.

IBM® may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.
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/VM.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

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 [IBM copyright and trademark information - United States](http://www.ibm.com/legal/us/en/copytrade.shtml).

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.
Glossary

For a list of z/VM terms and their definitions, see [z/VM: Glossary](#).

The z/VM glossary is also available through the online z/VM HELP Facility. For example, to display the definition of the term “dedicated device”, issue the following HELP command:

```
help glossary dedicated device
```

While you are in the glossary help file, you can do additional searches:

- To display the definition of a new term, type a new HELP command on the command line:

  ```
  help glossary newterm
  ```

  This command opens a new help file inside the previous help file. You can repeat this process many times. The status area in the lower right corner of the screen shows how many help files you have open. To close the current file, press the Quit key (PF3/F3). To exit from the HELP Facility, press the Return key (PF4/F4).

- To search for a word, phrase, or character string, type it on the command line and press the Clocate key (PF5/F5). To find other occurrences, press the key multiple times.

  The Clocate function searches from the current location to the end of the file. It does not wrap. To search the whole file, press the Top key (PF2/F2) to go to the top of the file before using Clocate.
See the following publications for additional information about z/VM. For abstracts of the z/VM publications, see z/VM: General Information.

Where to Get z/VM Information

z/VM product information is available from the following sources:
- z/VM V6R2 Information Center (publib.boulder.ibm.com/infocenter/zvm/v6r2/)
- IBM: z/VM Internet Library (www.ibm.com/vm/library/)
- IBM Publications Center (www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss)
- IBM Online Library: z/VM Collection, SK5T-7054

z/VM Base Library

Overview
- z/VM: General Information GC24-6193
- z/VM: Glossary GC24-6195
- z/VM: License Information GC24-6200

Installation, Migration, and Service
- z/VM: Installation Guide GC24-6246
- z/VM: Migration Guide GC24-6201
- z/VM: Service Guide GC24-6247
- z/VM: VMSES/E Introduction and Reference GC24-6243

Planning and Administration
- z/VM: CMS File Pool Planning, Administration, and Operation SC24-6167
- z/VM: CMS Planning and Administration SC24-6171
- z/VM: Connectivity SC24-6174
- z/VM: CP Planning and Administration SC24-6178
- z/VM: Getting Started with Linux on System z SC24-6194
- z/VM: Group Control System SC24-6196
- z/VM: I/O Configuration SC24-6198
- z/VM: Running Guest Operating Systems SC24-6228

Customization and Tuning
- z/VM: CP Exit Customization SC24-6166
- z/VM: Performance SC24-6208

Operation and Use
- z/VM: CMS Commands and Utilities Reference SC24-6166
- z/VM: CMS Pipelines Reference SC24-6169
- z/VM: CMS Pipelines User’s Guide SC24-6170
- z/VM: CMS Primer SC24-6172
- z/VM: CMS User’s Guide SC24-6173
- z/VM: CP Commands and Utilities Reference SC24-6175
- z/VM: System Operation SC24-6233
- z/VM: TCP/IP User’s Guide SC24-6240
- z/VM: Virtual Machine Operation SC24-6241
- z/VM: XEDIT Commands and Macros Reference SC24-6244
- z/VM: XEDIT User’s Guide SC24-6245
- CMS/TSO Pipelines: Author’s Edition SL26-0018

Application Programming
- z/VM: CMS Application Development Guide SC24-6162
- z/VM: CMS Application Development Guide for Assembler SC24-6163
- z/VM: CMS Application Multitasking SC24-6164
- z/VM: CMS Callable Services Reference SC24-6165
- z/VM: CMS Macros and Functions Reference SC24-6168
- z/VM: CP Programming Services SC24-6179
- z/VM: CPI Communications User’s Guide SC24-6180

© Copyright IBM Corp. 2003, 2011
Diagnosis

- z/VM: CMS and REXX/VM Messages and Codes GC24-6161
- z/VM: CP Messages and Codes GC24-6177
- z/VM: Diagnosis Guide GC24-6187
- z/VM: Dump Viewing Facility GC24-6191
- z/VM: Other Components Messages and Codes GC24-6207
- z/VM: TCP/IP Diagnosis Guide GC24-6235
- z/VM: TCP/IP Messages and Codes GC24-6237
- z/VM: VM Dump Tool GC24-6242
- z/OS and z/VM: Hardware Configuration Definition Messages SC33-7986

z/VM Facilities and Features

Data Facility Storage Management Subsystem for VM

- z/VM: DFSMS/VM Customization SC24-6181
- z/VM: DFSMS/VM Diagnosis Guide GC24-6182
- z/VM: DFSMS/VM Messages and Codes GC24-6183
- z/VM: DFSMS/VM Planning Guide SC24-6184
- z/VM: DFSMS/VM Removable Media Services SC24-6185
- z/VM: DFSMS/VM Storage Administration SC24-6186

Directory Maintenance Facility for z/VM

- z/VM: Directory Maintenance Facility Commands Reference SC24-6188
- z/VM: Directory Maintenance Facility Messages GC24-6189
- z/VM: Directory Maintenance Facility Tailoring and Administration Guide SC24-6190

Open Systems Adapter/Support Facility

Prerequisite Products

Device Support Facilities

Environmental Record Editing and Printing Program
- Environmental Record Editing and Printing Program (EREP): Reference GC35-0152
- Environmental Record Editing and Printing Program (EREP): User's Guide GC35-0151

Additional publications

z/OS Language Environment
- z/OS: Language Environment Concepts Guide SA22-7567
- z/OS: Language Environment Programming Guide SA22-7561
- z/OS: Language Environment Programming Reference SA22-7562
- z/OS: Language Environment Debugging Guide GA22-7560
- z/OS: Language Environment Writing Interlanguage Communication Applications SA22-7563
- z/OS: Language Environment Run-Time Messages SA22-7566

Bibliography 135
Index

Special characters

__csplist macro 110
__osplist macro 110
__pcblist macro 110
__R1 macro 110
@DELETE service routine for preinitialization
  components of 51
  return/reason codes for 51
@EXCEPRTN service routine for preinitialization
  return/reason codes for 54
@FREESTORE service routine for preinitialization
  return/reason codes for 52
@GETSTORE service routine for preinitialization
  return/reason codes for 52
@LOAD service routine for preinitialization
  components of 50
  return/reason codes for 50
@MSGRTN service routine for preinitialization
  components of 55
  return/reason codes for 55
#pragma directives
  See pragma
c89 utility
  build object modules 23
  interface to the linkage editor 23

A

abend codes
  abend 2106, reason code 7205 123
  abend 4093, reason code 60 115
  CEEAUE_RETC field of CEEBXITA and 45
abends
  CICS
    assembler user exit and EXEC CICS ABEND 46
  dump, requesting in CEEBXITA assembler user exit 47
  nested enclaves and
    created by C system() 67
    created by CMSCALL 63
    created by SVC LINK 63
abnormal termination
  See abends
ACCEPT statement 37
addressing mode
  See AMODE
AMODE
  (24 | 31 | ANY) LOAD command option 10
  for CEEBXITA user exit 44
  specifying addressing mode on CMS LOAD command 10
application
  See also enclave
  building using c89 23
argc parameter for C
  C parameter passing styles and 112
argument
  list format
    EXECOPS run-time option and 111, 112
    how interactions of EXECOPS and PLIST
      run-time options affect 112
    PLIST run-time option and 111
  passing
    C parameter passing for operating systems and
      subsystems 109
    specifying to an invoked routine which format to
      expect (C) 111
  argv parameter for C
    C parameter passing styles and 112
ASSEMBLE file 29
assembler language
  COBOL parameter list format 113
  system programming C considerations 127
  user exit
    See CEEBXITA assembler user exit
AUTO | NOAUTO prelinker option 97
automatic library call processing
  See library call processing

B

binder interface
  c89 utility 23
building freestanding applications
  including alternate initialization routines for 121
  VM/CMS 121
BYVALUE compiler option
  required if SYSTEM(CICS) specified 113

C

#pragmamas
  See pragma
building system exit routines 123
examples
  freestanding VM routines 122
exit() function
  EDCXEXIT routine and 122
functions
  See main routine
  See subroutine
L-names
  See L-names
LONGNAME compiler option 95
NOSTART compiler option 121
OPTIMIZE(2) compiler option 123
parameter passing, for operating systems and
  subsystems
  PLIST and EXECOPS interactions 111
  styles 109
pragma
  See pragma
C (continued)
   prelinker
      See prelinker
   S-names
      See S-names
   stderr
      default destinations of 35
      interleaving output with other output 36
      redirecting output from 36
   system programming facilities
      See system programming facility, C
   C370LIB EXEC 115
   c89 utility
      See OpenExtensions, c89 utility
   CALL statement
      for PL/I
      See FETCH statement
   casting, when using R1 and osplist macros 110
   CC EXEC
      creating object library 115
   CEEAUE_A_AB_CODES
      description 48
   CEEAUE_A_CC_PLIST 47
   CEEAUE_A_OPTIONS 47
   CEEAUE_ABND 47
   CEEAUE_ABTERM 46
   CEEAUE_DUMP 47
   CEEAUE_FBCODE 48
   CEEAUE_FLAGS
      CEEAUE_ABND field of 47
      CEEAUE_ABTERM field of 46
      CEEAUE_DUMP field of 47
      CEEAUE_STEPS field of 47
      format 46
   CEEAUE_FUNC 45
   CEEAUE_LEN 45
   CEEAUE RETC
      description 45
      relationship to CEEAUE_ABND 45, 46
      relationship to CEEAUE_RSNC 46
   CEEAUE_RSNC
      description 46
      relationship to CEEAUE_ABND 46
      relationship to CEEAUE_RETCC 46
   CEEAUE_STEPS 47
   CEEAUE_USERWD 47
   CEEAUE_WORK 47
   CEEBIND T HLL user exit
      when invoked 43
   CEEBIXITA assembler user exit
      AMODE/RMODE considerations 44
      application-specific 41
      behavior of
         during enclave initialization 43
         during enclave termination 44
      functions 41
      installation-wide 41
   CEEBXITA assembler user exit (continued)
      interface to
         See also CEXIT control block
      diagram of 44
      modifications to, rules for making 44
      specifying run-time options in 47
      when invoked 43
      work area for 47
   CEECIXITA assembler user exit
      See CEEBXITA assembler user exit
   CEEFDUMP default dump file
   CEEBXITA assembler user exit and 44
   CEESTART
      LOAD command and 11
      specify RESET CEESTART for C main routines 9
   CEEOPT macro 25
   CEES transient data queue
      message handling and 35
   changes, summary xvii
   CICS
      See also EXEC CICS command
      COBOL parameter list formats 113
      PLIST and EXECOPS interactions 111
      SYSTEM setting 113
   CLISTs for TSO
      CPLINK 105
   CMOD EXEC
      C prelinker and 106
      syntax description 15
   CMS
      building freestanding applications 121, 127
      building persistent C environments 123
      building system exit routines 123
      CMOD 15, 17, 106
      COBOL parameter list formats 113
      CPLINK
         example using 106
         syntax description 105
      dynamically loaded routines
         search order for 8
         where installed 4
   EXECs
      See CMS, return code considerations
   LINKLOAD EXEC 17
   loading for
      basics 5
      FILEDEF command and 14
      GENMOD command and 12, 14
      INCLUDE command and 12
      LINKED command and 14
      methods of 5
      options 10
   OSRUN command
      See OSRUN command for CMS
   PLIST and EXECOPS interactions 111, 112
   prelinking for
      automatic library call processing 97
      invoking the prelinker 105
      prelinker input 96
      prelinking options 107
      return code considerations 27
CMS (continued)
running for
basics 5
GENMOD command and 12, 20
GLOBAL command and 6
LKED command and 20
START command and 18
SYSTEM setting 113
using system programming facilities 121, 127
where library routines are stored 3
CMSTPL SYSTEM setting 113

COBOL
non-CICS OS/VS COBOL programs supported in
single enclave only 62
parameter list formats 113
STOP RUN statement
CEEBXITA assembler user exit and 42
code packaging 3
command processor parameter list (CPPL)
PLIST, EXECOPS and 112
condition
nested 65
severity
CEEBXITA assembler user exit and 46
condition handling
nested enclaves
created by C system() 66
created by CMS/CMSCALL 63, 64
created by SVC LINK 63, 64
with a PL/I fetchable main 66, 68
constructed reentrancy
See prelinker
COPY file 29
CPLINK EXEC
example using 122
syntax description 105
CPPL (command processor parameter list)
See command processor parameter list (CPPL)
cross system product (CSP) 110
csplist macro 110
CEXIT control block
CEEAE_A_CC_PLIST field of 47
CEEAE_A_OPTIONS field of 47
CEEAE_FBCODE field of 48
CEEAE_FLAGS field of
CEEAE_DUMP field of 47
CEEAE_STEPS field of 47
format of the 46
CEEAE_FUNC field of 45
CEEAE_LEN field of 45
CEEAE_WORK field of 47

D
DELETE service routine for preinitialization
components of 51
return/reason codes for 51
dereferencing 110
diagnosis checklist 79
DISPLAY statement
default file for 37
dump
CEEBXITA assembler user exit and 47
dynamic routines 3

E
enclave
nested
created by C system() function 61, 65
created by CMS/CMSCALL 61, 62, 64
created by SVC LINK 62, 64
ever with a PL/I fetchable main routine 66, 68
termination
with abend 46
entry point
running default entry point under CMS 18
ESD map of defined and longnames 100
examples
CMOD EXEC 106
freestanding C MVS routine 122
freestanding C VM routine 121
GENMOD command for CMS 121
building freestanding VM routine 122
including alternate initialization routines 121
GLOBAL command 121
building freestanding routine 121
including alternate initialization routines
CMS 121
invoking the prelinker
from CMS 105
linking and running under CMS 5
LOAD command for CMS
building freestanding VM routine 122
including alternate initialization routines 123
loading files under CMS 10
relinking PL/I applications 4
using PL/I routine as nucleus extension 13
EXCEPRTN service routine for preinitialization
return/reason codes for 54
EXEC CICS command
ABEND 46
EXECOPS run-time option
CMS START command and 18, 19
interaction with PLIST run-time option, under
CMS 112
MVS argument list format and 111
EXECs for CMS
See CMS, return code considerations
EXECs, IBM-supplied
C370LIB 115
LINKLOAD 116
EXHIBIT for OS/VS COBOL
default output file of 37
no support for, under CICS 37
exit() function
system programming facilities and 122
extended parameter list for CMS 21
FETCH statement
fetchable main
discussion of 66, 68
reentrancy considerations of 67
FILEDEF command for CMS
during enclave initialization 44
example of 14, 15
SYSABEND PRINTER 47
SYSMDUMP PRINTER 47
SYSUDUMP PRINTER 47
using to relate a ddname to an I/O device 14
FILENAME option
TXTLIB command 115
fprintf function 36
freestanding application
alternate initialization routines for 121
building
VM 121, 123
FREESTORE service routine for preinitialization
return/reason codes for 52
freopen 36
GENMOD command for CMS
example 13, 121, 122
executing module produced by 20
link-editing process and 5, 12
LOAD command, LKED command and 19
syntax description 12
genxlt
EXEC 19
utility
CMS 19
GETSTORE service routine for preinitialization
return/reason codes for 52
global assembler user exit 41
GLOBAL command for CMS
alternate initialization routines and 121
freestanding C applications and 121
GENMOD command and 20
LOAD command and 6
START command and 18
global error table
See condition handling
header files
stdlib.h and the __R1 and __osplst macros 110
L-names
LIBRARY control statement and 103
mapping to S-names 104
RENAME control statement and 103
resolving undefined 97
unresolved 97
UPCASE prelink option and 107
library call processing
prelinker and 97
LIBRARY statement
prelinker and 103
link-editing
for TSO
basics of linking and running 4
linkage editor
function 3
LINKLOAD EXEC
#pragma linkage and 17
options 116
options for 17
syntax description 17
LKED command for CMS
element using 15
FILEDEF command and 14
LOAD command, GENMOD command and 19
LKED command for CMS (continued)
running module produced by 20
syntax description 14
LOAD command for CMS
alternate initialization routines and 121
C System Exit routines and 123
element using 9
freestanding applications and 122
GLOBAL commands and 19
options for 10
persistent C environment and 123
syntax description 9
using multiple times 12
LOAD service routine for preinitialization
components of 50
return/reason codes for 50
LOADLIB for CMS
LKED command and 15
running under CMS and 20, 21
search order of 8
LONGNAME compiler option 95
longname support 115

M
macro
__cspplist 110
__osplist 110
__pcklist 110
__R1 110
main routine
nested enclave considerations 61
map heading 118
mapping
L-names to S-names 104
member heading 118
message
directing to an I/O device 35
using in your application 35
message file
C stderr and stdout output and 36
CICS considerations 35
COBOL DISPLAY statement and 38
Language Environment's default destinations 34
nested enclave considerations 70
PL/I I/O statements 38
specifying ddname of 35
using CEEBLDTX to assemble 29
message handling
specifying ddname of message file 35
message module table 29
MSGFILE run-time option
default destinations under different operating systems 35
under OpenExtensions 35
MSGRTN service routine for preinitialization
components of 55
return/reason codes for 55

N
naming convention for object library members 116
nested enclave
See also enclave, nested
See SVC LINK
nonoverrideable 47
NOOE prelinker option 24
Notices 127
nucleus extension
Language Environment library routines and 4
routine search order in CMS running procedure 8

O
object library utility 115
adding object modules 115
deleting object modules 115
element 116
listing the contents 115
Object Library Utility 115
OE prelinker option 24
OpenExtensions 23, 26
building and running, basic 23
building C applications 23
building PL/I applications 26
c89 utility
-c option 23
-o option 23
-W option 24
default prelinker settings 24
forced prelinker settings 24
OE option 24
environments supported 23, 25
MSGFILE run-time option and 35
prelinking under 24
run-time options under 25
running
C applications 25, 26
from z/OS UNIX shell 25, 26
OPTIONS(BYADDR)
assembler calling PL/I under IMS 113
SYSTEM(CICS) and 113
OPTIONS(BYVALUE)
IMS considerations 113
SYSTEM(CICS) and 113
osplist macro 110
OSRUN command for CMS
GLOBAL command and 20
LKED command and 20
PLIST and EXECOPS run-time options and 112
PLIST run-time option and 111, 112
syntax description 20
overrideable/nonoverrideable 47

P
parameter
list format
effect of EXECOPS run-time option 111, 112
parameter (continued)
  list format (continued)
    how interaction of EXECOPS and PLIST run-time
    options affects 112
    PLIST run-time option and 111
    passing
    C passing styles 109
  pcblist macro 110
  persistent C environment 123

PL/I
  BYADDR  113
  BYVALUE
    must be specified if SYSTEM(IMS) or
    SYSTEM(CICS) specified  113
    nucleus extension, using PL/I routine as  13
    SYSTEM compiler option
    interactions with NOEXECOPS  113, 115

PLIST run-time option
  argument list format and 111, 112
  HOST setting and portability 111
  interaction with EXECOPS run-time option, under
  CMS  112

pragma
  #pragma linkage  17
  #pragma runopts
    affecting argument list format with 111, 112
  preinitialization facility  49

prelinker
  functions  96
    how it maps L-names to S-names  104
    how it resolves undefined symbols  97
    INCLUDE statement and  102
    input  96
    invoking
      for VM/CMS  105
  LIBRARY statement and 103
    prelinker options  107
  prelinker map  98
  RENAME statement and 103
    when it has to be used  95

preventive service planning (PSP) bucket  79
printf() function
  default destination  36
  interspersing messages into an application  35

process
  assembler user exit for termination of  45
  program
    building using c89  23
  PSP (preventive service planning) bucket  79

RENAMEx control statement
  how prelinkage utility maps L-names to
  S-names  104
  syntax and usage notes  103

RENT compiler option
  prelinker must be used when C source file compiled
  with  95
  resident routines  3

return code
  CEEAUE_RETC field of CEEBXITA and  45
  CMS considerations  27
    in user exits  45

RTLS run-time option
  with CEEAUE_A_OPTIONS output string  47
  run-time options
    EXECOPS — let run-time options be specified on
    command line
    See EXECOPS run-time option
    how nested enclaves get
    enclaves created by C system()  65
    enclaves created by CMSCALL  62
    enclaves created by SVC LINK  62
    in the user exit  44, 47
    MSGFILE — specify dname of diagnostic file
    See MSGFILE run-time option
    PLIST — specify format of C arguments
    See PLIST run-time option
    RTLS — modify search order when modules are
    loaded
    See RTLS run-time option
    TRAP — handle abends and program interrupts
    See TRAP run-time option

S

S-names
  prelinker and
    how L-names are mapped to S-names  104
    how S-names found in input are handled  96
    how unresolved S-names are handled  97

saved segments  8
  Language Environment library routines and  4
  routine search order in CMS running procedure  8

SCEELKED link library
  C system programming facility of C and  122
  CMS load procedures and  7, 15
    code packaging and  3

SCEERUN load library
  CMS load/run procedures and  20
    code packaging and  4

search order
  dynamically loaded routines for CMS  8
  severity
    of a condition
      CEEBXITA assembler user exit and  46
  signal()
    See condition handling
  standard streams  36

START command for CMS  18
  stderr
    default destinations of  35
subroutine
  restriction regarding nested enclaves 61
SVC LINK 62
symbol information 119
SYSABEND PRINTER 47
SYSIN 44
SYSDUMP PRINTER 47
SYSOUT
  CEEBXITA assembler user exit and 44
default destinations of MSGFILE run-time option 34
destination when inserting messages in your
application 38
system programming facility, C 121, 127
  building freestanding applications 121
  persistent C environments 123
  reentrant modules 122
  summary of functions 124
  system exit routines 123
  user-server environments 123
SYSUDUMP PRINTER 47

T
termination
  enclave
    as indicated in CEEAU E_ABND field of
    CEEAU E_FLAGS 46
    as indicated in CEEAU E_ABTERM field of
    CEEAU E_FLAGS 46
    CEEBXITA function codes for 45
  process
    CEEBXITA function code for 45
TRAP run-time option
  how CEEAU E_ABND is affected by 46
  nested enclaves and
    enclaves created by C system() 66
    enclaves with a C or assembler main; created by
    CMSCALL 63, 64
    enclaves with a COBOL main, created by
    SVC LINK 63, 64
    enclaves with a COBOL main, created by SVC
    CMSCALL 64
    enclaves with a COBOL main, created by SVC
    LINK 64
    enclaves with a PL/I fetchable main 66, 68
TXTLIB
  creating 115
TXTLIB command
  FILENAME option 115
TXTLIB for CMS 8, 14

U
user
  exit
    assembler 44
    for initialization 44
    for termination 44
    system exits in C Systems Programming
      Environment 123
      under CICS 45, 47
user (continued)
  return code
    See return code
user comments 119
user-server environment 123

W
writable static
  handled by prelinker 96
  writable static map 98

X
XITPTR 44