Contents

Figures ........................................ v
Tables ........................................... xiii

About this document .............................. xv
Using your documentation ........................ xvi
How to read syntax diagrams ........................ xvii
Symbols ............................................ xvii
Syntax items ........................................ xvii
Syntax examples .................................... xviii
z/OS information ................................... xix

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

Summary of changes ................................ xxiii
Summary of changes for Language Environment
for z/OS Version 2 Release 3 (V2R3) ............ xxiii
Summary of changes for Language Environment
for z/OS Version 2 Release 2 (V2R2) ............ xxiv
Summary of changes for Language Environment
for z/OS Version 2 Release 1 (V2R1) as updated
February, 2015 ..................................... xxiv
z/OS Version 2 Release 1 summary of changes xxiv

Part 1. Introduction to debugging in
Language Environment .......................... 1

Chapter 1. Preparing your routine for
debugging ......................................... 3
Setting compiler options .......................... 3
  XL C and XL C++ compiler options .......... 3
  COBOL compiler options ....................... 5
  Fortran compiler options ...................... 6
  PL/I compiler options ......................... 7
  Enterprise PL/I for z/OS compiler options .. 8
Using Language Environment runtime options .. 9
  Determining runtime options in effect ....... 10
  Understanding the HEAPZONES and HEAPCHK runtime options ............. 13
  Using the CLER CICS transaction to display and set runtime options .......... 13
Controlling storage allocation .................... 14
  Stack storage statistics ....................... 20
  Heap storage statistics ....................... 21
  HEAPPOOLS storage statistics ............... 23
Modifying condition handling behavior ........ 23
  Language Environment callable services ........ 23
  Language Environment runtime options ........ 24
  Customizing condition handlers .............. 26
  Invoking the assembler user exit .......... 26
  Establishing enclave termination behavior for unhandled conditions ....... 27
Using messages in your routine .................. 28
  C/C++ ........................................... 28
  COBOL .......................................... 28
  Fortran ......................................... 28
  PL/I ............................................. 29
Using condition information ...................... 29
  Using the feedback code parameter ........... 29
  Using the symbolic feedback code .......... 31

Chapter 2. Classifying errors .................... 33
Identifying problems in routines ................. 33
  Language Environment module names .......... 33
  Common errors in routines ..................... 33
Interpreting runtime messages .................... 35
  Message prefix .................................. 35
  Message number .................................. 36
  Severity code .................................... 36
  Message text .................................... 36
Understanding abend codes ........................ 36
  User abends ..................................... 36
  System abends .................................... 37
Using edcmtext to obtain information about ermm2 values ....................... 37
  Format ............................................ 37
  Description ...................................... 37
  Usage notes ..................................... 37
  Message returns .................................. 38
  Examples ........................................ 38
  Exit Values ..................................... 38

Chapter 3. Using Language Environment debugging facilities .......... 39
Debug tools ....................................... 39
  Language Environment dump service, CEE3DMP ....... 39
  Generating a Language Environment dump with CEE3DMP .................. 39
  Generating a Language Environment dump with TERMTHDACT ............... 42
  Generating a Language Environment dump with language-specific functions .... 46
Understanding the Language Environment dump ............ 47
Debugging with specific sections of the Language Environment dump .......... 66
Multiple enclave dumps ................................ 83
Generating a system dump .......................... 85
  Steps for generating a system dump in a batch runtime environment ........ 85
  Steps for generating a system dump in an IMS runtime environment .......... 85
  Steps for generating a system dump in a CICS runtime environment ........... 87
  Steps for generating a Language Environment U4039 abend .................. 87
  Steps for generating a system dump in a z/OS UNIX shell ..................... 88
Formatting and analyzing system dumps .......................... 89
Preparing to use the Language Environment support for IPCS ........................................... 89
Understanding Language Environment IPCS VERBEXIT = LEDATA ..................................... 90
Understanding the Language Environment IPCS VERBEXIT LEDATA output ................................ 93
Understanding the HEAP LEDATA output ......................... 114
Understanding the heap pools LEDATA output .................... 119
Understanding the heap pools trace LEDATA output .............. 124
Understanding the C/C++-specific LEDATA output ... 127
Understanding the COBOL-specific LEDATA output ............. 146
Understanding the PL/I-specific LEDATA output .................. 150
Formatting individual control blocks ............................... 156
Controlling access to CEEDUMPs and DYNDUMPs ............... 158
Requesting a Language Environment trace for debugging .......... 159
  Locating the trace dump ....................................... 160
  Using the Language Environment trace table format in a dump report .................................. 161
  Understanding the trace table entry (TTE) ................. 161
  Sample dump for the trace table entry ........................ 168
Requesting a UNIX System Services syscall trace for debugging ...... 169

Part 2. Debugging language-specific routines ............... 171

Chapter 4. Debugging C/C++ routines .............................. 173
Debugging C/C++ programs .................................. 173
  Using the __amrc and __amrc2 structures to debug input/output ........................................ 173
  Using file I/O tracing to debug C/C++ file I/O problems ........................................ 180
  Displaying an error message with the perror() function ........................................ 180
  Using __errno2() to diagnose application problems ........................................ 180
Diagnosing DLL problems ..................................... 182
Using C/C++ listings ........................................ 182
Finding variables ............................................. 183
Generating a Language Environment dump of a C/C++ routine .......... 190
cdump() .................................................................. 191
csnap() .................................................................. 192
ctrace() .................................................................. 192
Sample C routine that calls cdump() ............................... 192
Sample C++ routine that generates a Language Environment dump ........................................ 193
Sample Language Environment dump with C/C++-specific information ........................................ 195
Finding C/C++ information in a Language Environment dump ........................................ 199
Sample Language Environment dump with XPLINK-specific information ........................................ 205
Finding XPLINK information in a Language Environment dump ........................................ 209
C/C++ contents of the Language Environment trace tables .............. 209
Debugging examples of C/C++ routines .................................. 215
  Divide-by-zero error ........................................... 215
  Calling a nonexistent non-XPLINK function ........................................ 220
  Calling a nonexistent XPLINK function ........................................ 223
Handling dumps written to the z/OS UNIX file system ............... 226
Multithreading consideration ...................................... 227
Understanding C/C++ heap information in storage reports .... 227
  Language Environment storage report with heap pools statistics ........................................ 228
  C function _uheapreport() storage report ........................................ 229
MEMCHECK VHM memory leak analysis tool .................................. 231
Invoking MEMCHECK VHM ....................................... 231
MEMCHECK VHM environment variables .................................. 232
MEMCHECK VHM report sample scenario ............................... 233
MEMCHECK VHM report examples .................................... 233

Chapter 5. Debugging COBOL programs ........................... 237
Determining the source of error ................................... 237
Tracing program logic ........................................... 237
Finding input/output errors ..................................... 238
Handling input/output errors ..................................... 238
Validating data (class test) ...................................... 238
Assessing switch problems ...................................... 238
Generating information about procedures ......................... 238
Using COBOL listings ........................................... 240
Generating a Language Environment dump of a COBOL program ........................................ 240
  COBOL program that calls another COBOL program ........................................ 240
  COBOL program that calls the Language Environment CEE3DMP callable service ................. 241
Sample Language Environment dump with COBOL-specific information ........................................ 242
Finding COBOL information in a dump .................................. 244
Debugging example COBOL programs .................................. 248
  Subscript range error ........................................... 248
  Calling a nonexistent subroutine .................................. 250
  Divide-by-zero error ........................................... 254

Chapter 6. Debugging Fortran routines .............................. 259
Determining the source of errors in Fortran routines ................... 259
Using Fortran compiler listings .................................... 259
Generating a Language Environment dump of a Fortran routine ........................................ 261
  DUMP/PDUMP subroutines ...................................... 262
  CDUMP/CPDUMP subroutines .................................... 263
  SDUMP subroutine ............................................. 263
Finding Fortran information in a Language Environment dump ........................................ 266
Examples of debugging Fortran routines .................................. 268
  Calling a nonexistent routine .................................... 268
  Divide-by-zero error ........................................... 270
Chapter 12. Using Language Environment AMODE 64 debugging facilities ........................................... 369
Debugging tools .................................................. 369
Language Environment dumps ................................. 369
Generating a Language Environment dump with TERMTHDACT ........................................... 369
Generating a Language Environment dump with language-specific functions ......................... 371
Understanding the Language Environment dump .............................................................. 371
Generating a system dump ...................................... 386
Steps for generating a system dump in a batch runtime environment ..................................... 387
Steps for generating a system dump in a z/OS UNIX shell .................................................. 387
Formatting and analyzing system dumps .............................................................. 388
Preparing to use the Language Environment support for IPCS ........................................... 388
Understanding Language Environment IPCS VERBEXIT – LEDATA ........................................... 389
Understanding the Language Environment IPCS VERBEXIT LEDATA output ..................... 393
Understanding the HEAP LEDATA output .................................................................. 408
Understanding the heap pool LEDATA output .................................................................. 418
Understanding the heap pools trace LEDATA output ........................................................ 424
Understanding the C/C++-specific LEDATA output ......................................................... 428
C/C++-specific sections of the LEDATA output .................................................................. 438
Understanding the PL/I-specific LEDATA output .................................................................. 439
PL/I-specific sections of the LEDATA output .................................................................. 445
Understanding the AUTH LEDATA output .................................................................. 446
Sections of the AUTH LEDATA VERBEXIT formatted output ........................................... 451
Formatting individual control blocks ........................................................................... 452
Requesting a Language Environment trace for debugging ........................................... 454
Locating the trace dump ................................................................................................. 455
Using the Language Environment trace table format in a dump report .................................. 456
Understanding the trace table entry (TTE) ........................................................................ 456
Sample dump for the trace table entry ........................................................................... 462
Requesting a UNIX System Services syscall trace for debugging ....................................... 463

Chapter 13. Debugging AMODE 64 C/C++ routines .............................................................. 465
Debugging C/C++ programs ......................................................................................... 465
Using the __amrc and __amrc2 structures to debug input/output ..................................... 466
Using file I/O tracing to debug C/C++ file I/O problems .................................................. 472
Displaying an error message with the perror() function .................................................. 473
Using __errno2() to diagnose application problems .................................................... 473
Using C/C++ listings ........................................................................................................ 475
Finding variables ........................................................................................................... 475
Generating a Language Environment dump of a C/C++ routine ......................................... 478
cdump() ......................................................................................................................... 479
csnap() ......................................................................................................................... 479
ctrace() ......................................................................................................................... 479
Sample C routine that calls cdump() ........................................................................... 479
Sample C++ routine that generates a Language Environment dump .................................. 481
Sample Language Environment dump with C/C++-specific information .......................... 482
C/C++ contents of the Language Environment trace tables ................................................ 487
Debugging examples of C/C++ routines .............................................................................. 489
Divide-by-zero error ......................................................................................................... 489
Calling a nonexistent function ....................................................................................... 496
Handling dumps written to the z/OS UNIX file system .................................................... 502
Multithreading consideration ......................................................................................... 503
Understanding C/C++ heap information in storage reports .............................................. 503
Language Environment storage report with heap pools statistics .................................... 503
C function __uheapreport() storage report ..................................................................... 506

Part 4. Appendixes .............................................................................................................. 509

Appendix A. Diagnosing problems with Language Environment ........................................... 511
Diagnosis checklist ............................................................................................................ 511
Locating the name of the failing routine for a non-XPLINK application ............................. 512
Searching the IBM Software Support Database ................................................................ 515
Preparing documentation for an authorized program analysis report (APAR) ..................... 515

Appendix B. Accessibility .................................................................................................... 519
Accessibility features ........................................................................................................ 519
Consult assistive technologies ......................................................................................... 519
Keyboard navigation of the user interface ....................................................................... 519
Dotted decimal syntax diagrams .................................................................................... 519

Notices ............................................................................................................................... 523
Terms and conditions for product documentation ................................................................ 525
IBM Online Privacy Statement ....................................................................................... 526
Policy for unsupported hardware ....................................................................................... 526
Minimum supported hardware ......................................................................................... 526
Programming interface information .................................................................................. 527
Trademarks ........................................................................................................................ 527

Index ................................................................................................................................. 529
Figures

1. Options report example produced by runtime option RPTOPTS(ON) .................. 12
2. Storage report produced by runtime option RPTSTG(ON) .................. 15
3. Storage report produced by runtime option RPTSTG(ON) (continued) .................. 16
4. Storage report produced by RPTSTG(ON) with XPLINK .................. 17
5. Storage report produced by RPTSTG(ON) with XPLINK (continued) .................. 18
6. Storage report produced by RPTSTG(ON) with XPLINK (continued) .................. 19
7. Language Environment condition token 30
8. The C program CELSAMP 48
9. The C program CELSAMP (continued) 49
10. The C program CELSAMP (continued) 50
11. The C DLL CELDLL 51
12. Upward-growing (non-XPLINK) stack frame format .......................... 67
13. Downward-growing (XPLINK) stack frame format .......................... 68
14. Condition information block (Part A) ........................................ 79
15. Condition information block (Part B) ........................................ 80
16. Machine state information block ........................................ 82
17. Language Environment dump of multiple enclaves .............................. 84
18. Example of formatted output from LEDATA VERBEXIT (Part 1 of 18) ................. 94
19. Example of formatted output from LEDATA VERBEXIT (Part 2 of 18) ................. 95
20. Example of formatted output from LEDATA VERBEXIT (Part 3 of 18) ................. 96
21. Example of formatted output from LEDATA VERBEXIT (Part 4 of 18) ................. 97
22. Example of formatted output from LEDATA VERBEXIT (Part 5 of 18) ................. 98
23. Example of formatted output from LEDATA VERBEXIT (Part 6 of 18) ................. 99
24. Example of formatted output from LEDATA VERBEXIT (Part 7 of 18) ................. 100
25. Example of formatted output from LEDATA VERBEXIT (Part 8 of 18) ................. 101
26. Example of formatted output from LEDATA VERBEXIT (Part 11 of 18) ................. 102
27. Example of formatted output from LEDATA VERBEXIT (Part 12 of 18) ................. 103
28. Example of formatted output from LEDATA VERBEXIT (Part 13 of 18) ................. 104
29. Example of formatted output from LEDATA VERBEXIT (Part 14 of 18) ................. 105
30. Example of formatted output from LEDATA VERBEXIT (Part 15 of 18) ................. 106
31. Example of formatted output from LEDATA VERBEXIT (Part 16 of 18) ................. 107
32. Example of formatted output from LEDATA VERBEXIT (Part 17 of 18) ................. 108
33. Example of formatted output from LEDATA VERBEXIT (Part 18 of 18) ................. 109
34. CAA formatted by the CBFORMAT IPCS command ............................. 157
35. Format of the trace table entry ........................................ 161
36. __amrc structure ........................................ 174
37. __amrc2 structure ........................................ 175
38. Example of a routine using perror() ..................................... 180
39. Example of a routine using __errno2() .................................... 181
40. Sample output of a routine using __errno2() ................................ 181
41. Example of a routine using _EDC_ADD_ERRN02 ................................ 181
42. Sample output of a routine using _EDC_ADD_ERRN02 ................................ 181
43. Example of a routine using __errno2() in combination with __errno2() ............ 182
44. Sample output of routine using __errno2() in combination with __errno2() ............ 182
45. Writable static map produced by prelinker ..................................... 185
46. Location of NORENT static variable in storage .................................... 185
47. Writable static map produced by prelinker ..................................... 186
48. Location of NORENT static variable in storage .................................... 187
49. Example code for parameter variable ........................................ 187
50. Example code for parameter variable ........................................ 188
51. Partial storage offset listing ........................................ 188
52. Example code for structure variable ........................................ 189
53. Example of aggregate map ........................................ 189
54. Writable static map produced by prelinker ..................................... 190
55. Fetched module for C routine ........................................ 193
56. Example C++ routine with protection exception generating a dump ............... 194
57. Template file STACK.C ........................................ 194
58. Header file STACK.h ........................................ 194
59. Memory file control block ........................................ 202
60. Registers on entry to CEE3DMP ........................................ 202
61. Parameters, registers, and variables for active routines ............................ 203
62. Condition information for active routines ..................................... 203
63. Registers on entry to CEE3DMP ........................................ 204
64. Parameters, registers, and variables for active routines ............................ 204
65. Condition information for active routines ..................................... 205
66. Sample XPLINK-assembled program (trnmain) which calls a NOXPLINK-assembled program ........................................ 206
67. Sample NOXPLINK-assembled program (trnlink) which calls an XPLINK-assembled program ........................................ 206
68. Trace table with trace table entry types 7 and 8 ..................................... 214
69. C routine with a divide-by-zero error ........................................ 216
70. Sections of the dump from example C/C++ routine (divide-by-zero error) ............ 217
71. Pseudo assembly listing (C/C++ routine divide-by-zero error) ................. 218
Tables

1. How to use z/OS Language Environment publications ........................................... xvi
2. Syntax examples ......................................................... xviii
3. TEST suboptions to simplify debugging ......................................... 3
4. C/C++ compiler options to simplify runtime debugging ......................... 4
5. COBOL compiler options for runtime debugging ....................................... 5
6. Fortran compiler options for runtime debugging ................................. 6
7. PL/I compiler options for debugging ............................................. 7
8. Enterprise PL/I for z/OS compiler options for debugging ....................... 8
9. Language Environment runtime options for debugging ......................... 9
10. Storage report fields that display stack storage statistics .......................... 20
11. Storage report fields that display heap storage statistics ....................... 22
12. Callable services that modify condition handling ..................................... 24
13. Runtime options that modify condition handling ..................................... 24
14. Callable services that can convert condition tokens to routine variables, messages, or signaled conditions ................................................. 30
15. Common error symptoms, possible causes, and programmer responses .......... 34
16. CEE3DMP options .......................................................... 41
17. TERMTHDACT suboptions, level of information, and destinations ............. 43
18. Condition handling of 0Cx abends .................................................. 45
19. Contents of the Language Environment dump ....................................... 60
20. Description of CAA fields ..................................................... 69
21. Contents of the LEDATA VERBEXIT formatted output ......................... 109
22. Contents of the Heap report sections of LEDATA output .......................... 118
23. Contents of heap pools report sections of LEDATA output ........................ 123
24. Contents of heap pools trace section of LEDATA output .......................... 126
25. Contents of C/C++-specific sections of LEDATA output .......................... 145
26. Contents of COBOL-specific sections of LEDATA Output .......................... 148
27. Contents of COBOL-specific sections of LEDATA Output (Enterprise COBOL V5.1 and later releases) ............................................... 150
28. Contents of PL/I-specific sections of LEDATA output ............................... 155
29. Language Environment Control blocks that can be individually formatted .......... 157
30. TERMTHDACT runtime option settings and dump contents produced ........... 159
31. LE=1 entry records ........................................................... 162
32. LE=2 entry records ........................................................... 162
33. Format of the mutex/CV/latch records ............................................... 167
34. LE=8 entry records ........................................................... 168
35. LE=20 entry records ......................................................... 168
36. __last_op values and diagnosis information ......................................... 176
37. Finding the WSA base address ..................................................... 183
38. Contents of the COBOL sections of Language Environment ....................... 199
39. Contents of XPLINK information in a Language Environment dump ............ 209
40. Compiler-generated COBOL listings and their contents .......................... 240
41. Compiler-generated Fortran listings and their contents ........................... 261
42. Understanding the Language Environment traceback table ........................ 267
43. Compiler-generated PL/I for MVS & VM listings and their contents ............. 276
44. Typical comments in a PL/I for MVS & VM static storage listing ................. 279
45. Comments in a PL/I for MVS & VM object code listing ............................ 281
46. PL/I for MVS & VM mnemonics .................................................. 282
47. Compiler-generated PL/I listings and their contents ................................ 310
48. Finding data when Language Environment returns a nonzero return code .......... 345
49. Finding data when Language Environment abends internally ..................... 345
50. Finding data when Language Environment abends from an EXEC CICS command .... 346
51. Common error symptoms, possible causes, and programmer responses .......... 366
52. TERMTHDACT suboptions, level of information, and destinations ............. 369
53. Contents of the Language Environment dump - AMODE 64 ....................... 382
54. Contents of the Language Environment LEDATA VERBEXIT formatted output (AMODE 64) .................................................. 404
55. Contents of Heap report sections of the LEDATA output (AMODE 64) .......... 417
56. Contents of the heap pool report sections of the LEDATA output (AMODE 64) .... 424
57. Contents of a detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64) .................................................. 428
58. Contents of C/C++-specific sections of the LEDATA output (AMODE 64) ........ 438
59. Contents of PL/I-specific sections of the LEDATA output (AMODE 64) .......... 445
60. Contents of AUTH LEDATA VERBEXIT formatted output (AMODE 64) ........ 451
61. Language Environment control blocks that can be individually formatted .......... 453
62. Preinitialized Environments for Authorized Programs control blocks that can be individually formatted ................................. 454
63. TERMTHDACT runtime option settings and dump contents produced (AMODE 64) . . . 454
64. LE=1 entry records (AMODE 64) . . . . . . . 457
65. LE=2 entry records (AMODE 64) . . . . . . . 457
66. Format of the mutex/CV/latch records (AMODE 64). . . . . . . . . . . . . . . 462
67. LE=8 entry records (AMODE 64) . . . . . 462
68. __last_op values and diagnosis information (AMODE 64). . . . . . . . . . . . . 469
69. Problem resolution documentation requirements. . . . . . . . . . . . . . . . . 516
About this document

**z/OS Language Environment Debugging Guide** provides assistance with detecting, finding, and fixing programming errors that occur during run time under Language Environment®. It can help you establish a debugging process to analyze data and narrow the scope and location of where an error might have occurred. You can read about how to prepare a routine for debugging, how to classify errors, and how to use the debugging facilities Language Environment provides. Also included are chapters on debugging HLL-specific routines and routines that run under CICS®. Debugging for AMODE 64 applications is covered in separate chapters, corresponding to the topics and contents that were provided.

This book is intended for application programmers who are interested in techniques for debugging runtime programs. You should be familiar with:

- Language Environment
- Appropriate languages that use the accepted compilers
- Program storage concepts

This document supports z/OS (5650-ZOS).

IBM® z/OS® Language Environment (also called Language Environment) provides common services and language-specific routines in a single runtime environment for C, C/C++, COBOL, Fortran (z/OS only; no support for z/OS UNIX or CICS), PL/I, and assembler applications. It offers consistent and predictable results for language applications, independent of the language in which they are written.

Language Environment is the prerequisite runtime environment for applications that are generated with the following IBM compiler products:

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

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

Language Environment supports, but is not required for, an interactive debug tool for debugging applications in your native z/OS environment.
Debug Tool is also available as a stand-alone product as well as Debug Tool Utilities and Advanced Functions. For more information, see the IBM z/OS Debugger (developer.ibm.com/mainframe/products/ibm-zos-debugger).

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

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

For more information about VisualAge for Java, Enterprise Edition for OS/390, program number 5655-JAV, see the product documentation.

Using your documentation

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

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

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

Each publication helps you perform different tasks, some of which are listed in Table 1.

Table 1. How to use z/OS Language Environment publications

<table>
<thead>
<tr>
<th>To ...</th>
<th>Use ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluate Language Environment</td>
<td>z/OS Language Environment Concepts Guide</td>
</tr>
<tr>
<td>Plan for Language Environment</td>
<td>z/OS Language Environment Concepts Guide</td>
</tr>
<tr>
<td>Install Language Environment</td>
<td>z/OS Program Directory in the z/OS Internet library (<a href="http://www.ibm.com/systems/z/os/zos/library/bkserv">www.ibm.com/systems/z/os/zos/library/bkserv</a>)</td>
</tr>
<tr>
<td>Customize Language Environment</td>
<td>z/OS Language Environment Customization</td>
</tr>
<tr>
<td>Understand Language Environment program models and concepts</td>
<td>z/OS Language Environment Concepts Guide</td>
</tr>
<tr>
<td>Find syntax for Language Environment runtime options and callable services</td>
<td>z/OS Language Environment Programming Reference</td>
</tr>
<tr>
<td>Develop applications that run with Language Environment</td>
<td>z/OS Language Environment Programming Guide and your language programming guide</td>
</tr>
</tbody>
</table>
Table 1. How to use z/OS Language Environment publications (continued)

<table>
<thead>
<tr>
<th>To ...</th>
<th>Use ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug applications that run with Language Environment, diagnose problems with Language Environment</td>
<td>z/OS Language Environment Debugging Guide</td>
</tr>
<tr>
<td>Get details on runtime messages</td>
<td>z/OS Language Environment Runtime Messages</td>
</tr>
<tr>
<td>Develop interlanguage communication (ILC) applications</td>
<td>z/OS Language Environment Writing Interlanguage Communication Applications and your language programming guide</td>
</tr>
<tr>
<td>Migrate applications to Language Environment</td>
<td>z/OS Language Environment Runtime Application Migration Guide and the migration guide for each Language Environment-enabled language</td>
</tr>
</tbody>
</table>

How to read syntax diagrams

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

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

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

Symbols

The following symbols may be displayed in syntax diagrams:

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

Syntax items

Syntax diagrams contain many different items. Syntax items include:

- **Keywords** - a command name or any other literal information.
- **Variables** - variables are italicized, appear in lowercase, and represent the name of values you can supply.
- **Delimiters** - delimiters indicate the start or end of keywords, variables, or operators. For example, a left parenthesis is a delimiter.
- **Operators** - operators include add (+), subtract (-), multiply (*), divide (/), equal (=), and other mathematical operations that may need to be performed.
- **Fragment references** - a part of a syntax diagram, separated from the diagram to show greater detail.
- **Separators** - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.
Note: If a syntax diagram shows a character that is not alphanumeric (for example, parentheses, periods, commas, equal signs, a blank space), enter the character as part of the syntax.

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

**Item type**

**Definition**

**Required**

Required items are displayed on the main path of the horizontal line. You must specify these items.

**Optional**

Optional items are displayed below the main path of the horizontal line. You may choose one of the items in the stack.

**Default**

Default items are displayed above the main path of the horizontal line.

### Syntax examples

The following table provides syntax examples.

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required item.</td>
<td>Required items appear on the main path of the</td>
</tr>
<tr>
<td></td>
<td>horizontal line. You must specify these items.</td>
</tr>
<tr>
<td>Required choice.</td>
<td>A required choice (two or more items) appears in a</td>
</tr>
<tr>
<td></td>
<td>vertical stack on the main path of the horizontal</td>
</tr>
<tr>
<td></td>
<td>line. You must choose one of the items in the stack.</td>
</tr>
<tr>
<td>Optional item.</td>
<td>Optional items appear below the main path of the</td>
</tr>
<tr>
<td></td>
<td>horizontal line.</td>
</tr>
<tr>
<td>Optional choice.</td>
<td>An optional choice (two or more items) appears in a</td>
</tr>
<tr>
<td></td>
<td>vertical stack below the main path of the horizontal</td>
</tr>
<tr>
<td></td>
<td>line. You may choose one of the items in the stack.</td>
</tr>
<tr>
<td>Default.</td>
<td>Default items appear above the main path of the</td>
</tr>
<tr>
<td></td>
<td>horizontal line. The remaining items (required or</td>
</tr>
<tr>
<td></td>
<td>optional) appear on (required) or below (optional)</td>
</tr>
<tr>
<td></td>
<td>the main path of the horizontal line. The following</td>
</tr>
<tr>
<td></td>
<td>example displays a default with optional items.</td>
</tr>
<tr>
<td>Variable.</td>
<td>Variables appear in lowercase italics. They</td>
</tr>
<tr>
<td></td>
<td>represent names or values.</td>
</tr>
</tbody>
</table>
### Table 2. Syntax examples (continued)

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeatable item.</td>
<td>![repeatable_item]</td>
</tr>
</tbody>
</table>

An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.

A character within the arrow means you must separate repeated items with that character.

An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.

Fragment.

The fragment symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.

```
fragment:
  ,required_choice1
  ,required_choice2
    ,default_choice
    ,optional_choice
```

---

### z/OS information

This information explains how z/OS references information in other documents and on the web.

When possible, this information uses cross document links that go directly to the topic in reference using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS, see [z/OS Information Roadmap](https://www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

To find the complete z/OS library, go to [IBM Knowledge Center](https://www.ibm.com/support/knowledgecenter/ssltbw/welcome)
How to send your comments to IBM

We appreciate your input on this documentation. Please provide us with any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

Use one of the following methods to send your comments:

**Important:** If your comment regards a technical problem, see instead “If you have a technical problem.”

- Send an email to mhvrcfs@us.ibm.com
- Send an email from the Contact z/OS web page (www.ibm.com/systems/z/os/zos/webqs.html)

Include the following information:
- Your name and address
- Your email address
- Your phone or fax number
- The publication title and order number: z/OS Language Environment Debugging Guide GA32-0908-30
- The topic and page number or URL of the specific information to which your comment relates
- The text of your comment.

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

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

If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one or more of the following actions:

- Visit the IBM Support Portal (support.ibm.com)
- Contact your IBM service representative.
- Call IBM technical support.
Summary of changes

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

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

The most recent updates are listed at the top of the section.

New

Because APAR PI91583 added support for lengths that are longer than 7 bytes, the service portion of Traceback for both CEEDUMP and LEDATA Verbexit was updated. See “Sections of the Language Environment dump” on page 59 and “Sections of the Language Environment LEDATA VERBEXIT formatted output” on page 109.

Support was provided for system times beyond 2038/2042. The following sections contain new information:

- Chapter 3, “Using Language Environment debugging facilities,” on page 39
  - “Understanding the C/C++-specific LEDATA output” on page 127
- Chapter 12, “Using Language Environment AMODE 64 debugging facilities,” on page 369
  - “Understanding the C/C++-specific LEDATA output” on page 428

Support added for stack guard. The following section contains new information for this support:

- “Common Anchor Area” on page 68

Language Environment now enforces the same security rules that are enforced by ABEND dump processing. For more information, see the new section “Controlling access to CEEDUMP’s and DYNDUMP’s” on page 158.

Added support to allow vector applications to run under ERTLI CICS. The following section contains new information:

- “Common Anchor Area” on page 68

Changed

Various updates were made to replace IBM Debug Tool for z/OS with IBM z/OS Debugger.
Summary of changes for Language Environment for z/OS Version 2 Release 2 (V2R2)

Changed

The following report headers were updated for V2R2:

- The options report header  
- The storage report header  
- The options report for AMODE 64 applications  
- The storage report for AMODE 64 applications

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

New

- Support was added for vectors. The following topics contain new information for this support:
  - “Understanding the Language Environment dump” on page 47
  - “Vector registers” on page 203
  - Chapter 5, “Debugging COBOL programs,” on page 237
  - “Divide-by-zero error” on page 334
  - “Understanding the Language Environment IPCS VERBEXIT LEDATA output” on page 393
  - “Divide-by-zero error” on page 489

z/OS Version 2 Release 1 summary of changes

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

- z/OS Migration
- z/OS Planning for Installation
- z/OS Summary of Message and Interface Changes
- z/OS Introduction and Release Guide
Part 1. Introduction to debugging in Language Environment

This part provides information about options and features you can use to prepare your routine for debugging. It describes some common errors that occur in routines and provides methods of generating dumps to help you get the information you need to debug your routine.
Chapter 1. Preparing your routine for debugging

This chapter describes options and features that you can use to prepare your routine for debugging. The following topics are covered:
- Compiler options for C, C++, COBOL, Fortran, and PL/I
- Language Environment runtime options
- Use of storage in routines
- Options for modifying condition handling
- Assembler user exits
- Enclave termination behavior
- User-created messages
- Language Environment feedback codes and condition tokens

Setting compiler options

The following sections discuss language-specific compiler options important to debugging routines in Language Environment. These sections cover only the compiler options that are important to debugging. For a complete list of compiler options, see the appropriate HLL publications.

The use of some compiler options (such as TEST) can affect the performance of your routine. You must set these options before you compile. In some cases, you might need to remove the option and recompile your routine before delivering your application.

XL C and XL C++ compiler options

When using XL C, set the TEST(ALL) suboption; this is equivalent to specifying TEST(LINE,BLOCK,PATH,SYM,HOOK). For XL C++, the option TEST is equivalent to TEST(HOOK). Table 3 lists the TEST suboptions that you can use to simplify runtime debugging.

<table>
<thead>
<tr>
<th>Suboption name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>Sets all of the TEST suboptions.</td>
</tr>
<tr>
<td>BLOCK</td>
<td>Generates symbol information for nested blocks.</td>
</tr>
<tr>
<td>HOOK</td>
<td>Generates all possible hooks. For details on this suboption, see z/OS XL C/C++ User’s Guide.</td>
</tr>
<tr>
<td>LINE</td>
<td>Generates line number hooks and allows a debugging tool to generate a symbolic dump.</td>
</tr>
<tr>
<td>PATH</td>
<td>Generates hooks at all path points; for example, hooks are inserted at if-then-else points before a function call and after a function call.</td>
</tr>
</tbody>
</table>
Table 3. TEST suboptions to simplify debugging (continued)

<table>
<thead>
<tr>
<th>Suboption name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM</td>
<td>Generates symbol table information and enables Language Environment to generate a dump at run time. When you specify SYM, you also get the value and type of variables displayed in the Local Variables section of the dump. For example, if in block 4 the variable x is a signed integer of 12, and in block 2 the variable x is a signed integer of 1, the following output appears in the Local Variables section of the dump:</td>
</tr>
<tr>
<td></td>
<td>%BLOCK4:&gt;x signed int 12</td>
</tr>
<tr>
<td></td>
<td>%BLOCK2:&gt;x signed int 1</td>
</tr>
<tr>
<td></td>
<td>If a nonzero optimization level is used, variables do not appear in the dump.</td>
</tr>
</tbody>
</table>

You can use the C/C++ compiler options shown in Table 4 to make runtime debugging easier. For a detailed explanation of these options, see z/OS XL C/C++ User’s Guide.

Table 4. C/C++ compiler options to simplify runtime debugging

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGGREGATE (C)</td>
<td>Specifies that a layout for struct and union type variables appear in the listing.</td>
</tr>
<tr>
<td>ATTRIBUTE (C++)</td>
<td>For XL C++ compile, cross reference listing with attribute information. If XREF is specified, the listing also contains reference, definition and modification information.</td>
</tr>
<tr>
<td>CHECKOUT (C)</td>
<td>Provides informational messages indicating possible programming errors.</td>
</tr>
<tr>
<td>EVENTS</td>
<td>Produces an events file that contains error information and source file statistics.</td>
</tr>
<tr>
<td>EXPMAC</td>
<td>Macro expansions with the original source.</td>
</tr>
<tr>
<td>FLAG</td>
<td>Specifies the minimum severity level that is tolerated.</td>
</tr>
<tr>
<td>GONUMBER</td>
<td>Generates line number tables corresponding to the input source file. This option is turned on when the TEST option is used. This option is needed to show statement numbers in dump output.</td>
</tr>
<tr>
<td>INFO (C++)</td>
<td>Indication of possible programming errors.</td>
</tr>
<tr>
<td>INLINE</td>
<td>Inline Summary and Detailed Call Structure Reports. (Specify with the REPORT sub-option).</td>
</tr>
<tr>
<td>INLRPT</td>
<td>Generates a report on status of functions that were inlined. The OPTIMIZE option must also be specified.</td>
</tr>
<tr>
<td>LIST</td>
<td>Listing of the pseudo-assembly listing produced by the compiler.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>Displays the offset addresses relative to the entry point of each function.</td>
</tr>
<tr>
<td>PHASEID</td>
<td>Causes each compiler module (phase) to issue an informational message which identifies the compiler phase module name, product identifier, and build level.</td>
</tr>
<tr>
<td>PPONLY</td>
<td>Completely expanded z/OS XL C, or z/OS XL C++ source code, by activating the preprocessor (PP) only. The output shows, for example, all the &quot;#include&quot; and &quot;#define&quot; directives.</td>
</tr>
<tr>
<td>SERVICE</td>
<td>Places a string in the object module, which is displayed in the traceback if the application fails abnormally.</td>
</tr>
</tbody>
</table>
Table 4. C/C++ compiler options to simplify runtime debugging (continued)

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOWINC</td>
<td>All included text in the listing.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Includes source input statements and diagnostic messages in the listing.</td>
</tr>
<tr>
<td>TERMINAL</td>
<td>Directs all error messages from the compiler to the terminal. If not specified, this is the default.</td>
</tr>
<tr>
<td>TEST</td>
<td>Generates information for debugging interface. This also generates symbol tables needed for symbolic variables in the dump.</td>
</tr>
<tr>
<td>XPLINK (BACKCHAIN)</td>
<td>Generates a prolog that saves redundant information in the calling function's stack frame.</td>
</tr>
<tr>
<td>XPLINK (STOREARGS)</td>
<td>Generates code to store arguments that are normally passed in registers, into the argument area.</td>
</tr>
<tr>
<td>XREF</td>
<td>For XL C compile, cross reference listing with reference, definition, and modification information. For XL C++ compile, cross reference listing with reference, definition, and modification information. If you specify ATTRIBUTE, the listing also contains attribute information.</td>
</tr>
</tbody>
</table>

See the Inter-procedural Analysis chapter in the [z/OS XL C/C++ Programming Guide](www.ibm.com/support/docview.wss?uid=swg27036733) for an overview and more details about Inter-procedural Analysis.

**COBOL compiler options**

When using COBOL V4R2 and prior releases, set the SYM suboption of the TEST compiler option. The SYM suboption of TEST causes the compiler to add debugging information into the object program to resolve user names in the routine and to generate a symbolic dump of the DATA DIVISION. With this suboption specified, statement numbers will also be used in the dump output along with offset values.

When using COBOL V5R1 and later releases, instead of setting the SYM suboption, set the DWARF suboption of the TEST compiler option. This has the same effect as the SYM option above concerning debug information in the object program.

To simplify debugging, use the NOOPTIMIZE compiler option. Program optimization can change the location of parameters and instructions in the dump output.

You can use the COBOL compiler options shown in Table 5 to prepare your program for runtime debugging. For more detail on these options and functions, see the appropriate programming guide in the Enterprise COBOL for z/OS library [www.ibm.com/support/docview.wss?uid=swg27036733](www.ibm.com/support/docview.wss?uid=swg27036733).

Table 5. COBOL compiler options for runtime debugging

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST</td>
<td>Produces a listing of the assembler expansion of your source code and global tables, literal pools, information about working storage, and size of routine’s working storage.</td>
</tr>
<tr>
<td>MAP</td>
<td>Produces lists of items in data division including a data division map, global tables, literal pools, a nested program structure map, and attributes.</td>
</tr>
</tbody>
</table>
Table 5. COBOL compiler options for runtime debugging (continued)

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET</td>
<td>Produces a condensed PROCEDURE DIVISION listing containing line numbers, statement references, and location of the first instruction generated for each statement.</td>
</tr>
<tr>
<td>OUTDD</td>
<td>Specifies the destination of DISPLAY statement messages.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Produces a listing of your source program with any statements embedded by PROCESS, COPY, or BASIS statements.</td>
</tr>
<tr>
<td>TEST</td>
<td>Produces object code that can run with a debugging tool, or adds information to the object program to produce formatted dumps. With or without any suboptions, this option forces the OBJECT option. When specified with any of the hook-location suboption values except NONE, this option forces the NOOPTIMIZE option. DWARF suboption includes statement numbers in the Language Environment dump and produces a symbolic dump. <strong>Note:</strong> For COBOL V4R2 and prior releases, use the SYM suboption instead of DWARF.</td>
</tr>
<tr>
<td>VBREF</td>
<td>Produces a cross-reference of all verb types used in the source program and a summary of how many times each verb is used.</td>
</tr>
<tr>
<td>XREF</td>
<td>Creates a sorted cross-reference listing.</td>
</tr>
</tbody>
</table>

**Fortran compiler options**

You can use these Fortran compiler options shown in Table 6 to prepare your program for runtime debugging. For more detail on these options and functions, see VS FORTRAN Version 2 Programming Guide for CMS and MVS or VS FORTRAN Version 2 Language and Library Reference.

Table 6. Fortran compiler options for runtime debugging

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIPS</td>
<td>Specifies if standard language flagging is to be performed. This is valuable if you want to write a program conforming to Fortran 77.</td>
</tr>
<tr>
<td>FLAG</td>
<td>Specifies the level of diagnostic messages to be written. I (Information), E (Error), W (Warning), or S (Severe). You can also use FLAG to suppress messages that are below a specified level. This is useful if you want to suppress information messages, for example.</td>
</tr>
<tr>
<td>GOSTMT</td>
<td>Specifies that statement numbers are included in the runtime messages and in the Language Environment dump.</td>
</tr>
<tr>
<td>ICA</td>
<td>Specifies if intercompilation analysis is to be performed, specifies the files containing intercompilation analysis information to be used or updated, and controls output from intercompilation analysis. Specify ICA when you have a group of programs and subprograms that you want to process together and you need to know if there are any conflicting external references, mismatched commons, and so on.</td>
</tr>
<tr>
<td>LIST</td>
<td>Specifies if the object module list is to be written. The LIST option lets you see the pseudo-assembly language code that is similar to what is actually generated.</td>
</tr>
<tr>
<td>MAP</td>
<td>Specifies if a table of source program variable names, named constants, and statement labels and their displacements is to be produced.</td>
</tr>
<tr>
<td>OPTIMIZE</td>
<td>Specifies the optimizing level to be used during compilation. If you are debugging your program, it is advisable to use NOOPTIMIZE.</td>
</tr>
</tbody>
</table>
Table 6. Fortran compiler options for runtime debugging (continued)

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDUMP</td>
<td>Specifies if dump information is to be generated.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Specifies if a source listing is to be produced.</td>
</tr>
<tr>
<td>SRCFLG</td>
<td>Controls insertion of error messages in the source listing. SRCFLG allows you to view error messages after the initial line of each source statement that caused the error, rather than at the end of the listing.</td>
</tr>
<tr>
<td>SXM</td>
<td>Formats SREF or MAP listing output to a 72-character width.</td>
</tr>
<tr>
<td>SYM</td>
<td>Invokes the production of SYM cards in the object text file. SYM cards contain location information for variables within a Fortran program.</td>
</tr>
<tr>
<td>TERMINAL</td>
<td>Specifies whether error messages and compiler diagnostics are to be written on the SYSTEM data set and whether a summary of messages for all the compilations is to be written at the end of the listing.</td>
</tr>
<tr>
<td>TEST</td>
<td>Specifies whether to override any optimization level above OPTIMIZE(0). This option adds runtime overhead.</td>
</tr>
<tr>
<td>TRMFLG</td>
<td>Specifies whether to display the initial line of source statements in error and their associated error messages at the terminal.</td>
</tr>
<tr>
<td>XREF</td>
<td>Creates a cross-reference listing.</td>
</tr>
<tr>
<td>VECTOR</td>
<td>Specifies whether to invoke the vectorization process. A vectorization report provides detailed information about the vectorization process.</td>
</tr>
</tbody>
</table>

PL/I compiler options

When using PL/I, specify the TEST compiler option to control the level of testing capability that are generated as part of the object code. Suboptions of the TEST option such as SYM, BLOCK, STMT, and PATH control the location of test hooks and specify whether or not a symbol table is generated. For more information about TEST, its suboptions, and the placement of test hooks, see the [IBM Enterprise PL/I for z/OS library](https://www.ibm.com/support/docview.wss?uid=swg27036735).

To simplify debugging and decrease compile time, set optimization to NOOPTIMIZE or OPTIMIZE(0). Higher optimization levels can change the location where parameters and instructions appear in the dump output.

You can use the compiler options listed in Table 7 to prepare PL/I routines for debugging. For more detail on PL/I compiler options, see the [IBM Enterprise PL/I for z/OS library](https://www.ibm.com/support/docview.wss?uid=swg27036735).

Table 7. PL/I compiler options for debugging

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGGREGATE</td>
<td>Specifies that a layout for arrays and major structures appears in the listing.</td>
</tr>
<tr>
<td>ESD</td>
<td>Includes the external symbol dictionary in the listing.</td>
</tr>
<tr>
<td>GONUMBER / GOSTMT</td>
<td>Tells the compiler to produce additional information specifying that line numbers from the source routine can be included in runtime messages and in the Language Environment dump.</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>Specifies that users can establish an ATTENTION ON-unit that gains control when an attention interrupt occurs.</td>
</tr>
</tbody>
</table>
Table 7. PL/I compiler options for debugging (continued)

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST</td>
<td>Produces a listing of the assembler expansion of source code and global tables, literal pools, information about working storage, and size of routine’s working storage.</td>
</tr>
<tr>
<td>LMESSAGE</td>
<td>Tells the compiler to produce runtime messages in a long form. If the cause of a runtime malfunction is a programmer’s understanding of language semantics, specifying LMESSAGE could better explain warnings or other information generated by the compiler.</td>
</tr>
<tr>
<td>MAP</td>
<td>Tells the compiler to produce tables showing how the variables are mapped in the static internal control section and in the stack frames, thus enabling static internal and automatic variables to be found in the Language Environment dump. If LIST is also specified, the MAP option also produces tables showing constants, control blocks, and INITIAL variable values.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>Specifies that the compiler prints a table of statement or line numbers for each procedure with their offset addresses relative to the primary entry point of the procedure.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Specifies that the compiler includes a listing of the source routine in the listing.</td>
</tr>
<tr>
<td>STORAGE</td>
<td>Includes a table of the main storage requirements for the object module in the listing.</td>
</tr>
<tr>
<td>TERMINAL</td>
<td>Specifies what parts of the compiler listing produced during compilation are directed to the terminal.</td>
</tr>
<tr>
<td>TEST</td>
<td>Specifies the level of testing capability that is generated as part of the object code. TEST also controls the location of test hooks and whether or not the symbol table is generated. Because the TEST option increases the size of the object code and can affect performance, limit the number and placement of hooks.</td>
</tr>
<tr>
<td>XREF and ATTRIBUTES</td>
<td>Creates a sorted cross-reference listing with attributes.</td>
</tr>
</tbody>
</table>

Enterprise PL/I for z/OS compiler options

Table 8 lists the Enterprise PL/I for z/OS compiler options that you can specify when preparing your Enterprise PL/I for z/OS routines for debugging. For further details on the Enterprise PL/I for z/OS compiler options, see the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

Table 8. Enterprise PL/I for z/OS compiler options for debugging

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGGREGATE</td>
<td>Specifies that a layout for arrays and major structures appears in the listing.</td>
</tr>
<tr>
<td>GONUMBER / GOSTMT</td>
<td>Tells the compiler to produce additional information specifying that line numbers from the source routine can be included in runtime messages and in the Language Environment dump.</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>Specifies that users can establish an ATTENTION ON-unit that gains control when an attention interrupt occurs.</td>
</tr>
<tr>
<td>LIST</td>
<td>Produces a listing of the assembler expansion of source code and global tables, literal pools, information about working storage, and size of routine’s working storage.</td>
</tr>
</tbody>
</table>
### Table 8. Enterprise PL/I for z/OS compiler options for debugging (continued)

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET</td>
<td>Displays the offset addresses relative to the entry point of each function.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Specifies that the compiler includes a listing of the source routine in the listing.</td>
</tr>
<tr>
<td>STORAGE</td>
<td>Includes a table of the main storage requirements for the object module in the listing.</td>
</tr>
<tr>
<td>TEST</td>
<td>Specifies the level of testing capability that is generated as part of the object code. TEST also controls the location of test hooks and whether or not the symbol table is generated. Because the TEST option increases the size of the object code and can affect performance, limit the number and placement of hooks.</td>
</tr>
<tr>
<td>XREF and ATTRIBUTES</td>
<td>Creates a sorted cross-reference listing with attributes.</td>
</tr>
</tbody>
</table>

---

### Using Language Environment runtime options

Several runtime options affect debugging in Language Environment. The TEST runtime option, for example, can be used with a debugging tool to specify the level of control the debugging tool has when the routine being initialized is started. The ABPERC, CHECK, DEPTHCONDLMT, DYNDUMP, ERRCOUNT, HEAPCHK, INTERRUPT, TERMTHDACT, TRACE, TRAP, and USRHDLR options affect condition handling. The ABTERMENC option affects how an application ends (that is, with an abend or with a return code and reason code) when an unhandled condition of severity 2 or greater occurs. Table 9 lists the Language Environment runtime options that affect debugging. For a more detailed discussion of these runtime options, see z/OS Language Environment Programming Reference.

### Table 9. Language Environment runtime options for debugging

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABPERC</td>
<td>Specifies that the indicated abend code bypasses the condition handler.</td>
</tr>
<tr>
<td>ABTERMENC</td>
<td>Specifies enclave termination behavior for an enclave ending with an unhandled condition of severity 2 or greater.</td>
</tr>
<tr>
<td>CEEDUMP</td>
<td>Specifies options to control the processing of the Language Environment dump report.</td>
</tr>
<tr>
<td>CHECK</td>
<td>Determines if runtime checking is performed.</td>
</tr>
<tr>
<td>NODEBUG</td>
<td>Controls the COBOL USE FOR DEBUGGING declarative.</td>
</tr>
<tr>
<td>DEPTHCONDLMT</td>
<td>Specifies the limit for the depth of nested synchronous conditions in user-written condition handlers. (Asynchronous signals do not affect DEPTHCONDLMT.)</td>
</tr>
<tr>
<td>DYNDUMP</td>
<td>Provides a way to obtain IPCS-readable dumps of user applications that would ordinarily be lost due to the absence of a SYSMDUMP, SYSUDUMP, or SYSABEND DD statement</td>
</tr>
<tr>
<td>ERRCOUNT</td>
<td>Specifies the number of synchronous conditions of severity 2 or greater tolerated. (Asynchronous signals do not affect ERRCOUNT.)</td>
</tr>
<tr>
<td>HEAPCHK</td>
<td>Determines if additional heap check tests are performed.</td>
</tr>
<tr>
<td>HEAPZONES</td>
<td>Activates user heap overlay toleration and checking.</td>
</tr>
</tbody>
</table>
### Table 9. Language Environment runtime options for debugging (continued)

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description</th>
</tr>
</thead>
</table>
| INFOMSGFILTER      | Filters user specified informational messages from the MSGFILE.  
  **Note:** Affects only those messages generated by Language Environment and any routine that calls CEEMSG. Other routines that write to the message file, such as CEEMOUT, do not have a filtering option. |
| INTERRUPT          | Causes Language Environment to recognize attention interrupts.                                                                                 |
| MSGFILE            | Specifies the dname of the Language Environment message file.                                                                                   |
| MSGQ               | Specifies the number of instance specific information (ISI) blocks that are allocated on a per-thread basis for use by an application.           
  Located within the Language Environment condition token is an ISI token. The ISI contains information used by the condition manager to identify and react to a specific occurrence of a condition. |
| PROFILE            | Controls the use of an optional profiler tool, which collects performance data for the running application. When this option is in effect, the profiler is loaded and the debugger cannot be loaded.  
  If the TEST option is in effect when PROFILE is specified, the profiler tool will not be loaded. |
| RPTOPTS            | Produces a report that shows the runtime options in effect; see “Determining runtime options in effect.”                                            |
| RPTSTG             | Generates a report of the storage used by an enclave; see “Controlling storage allocation” on page 14.                                             |
| STORAGE            | Specifies that Language Environment initializes all heap and stack storage to a user-specified value.                                            |
| TERMTHDACT         | Controls response when an enclave terminates due to an unhandled condition of severity 2 or greater.                                            |
| TEST               | Specifies the conditions under which a debugging tool assumes control.                                                                           |
| TRACE              | Activates Language Environment runtime library tracing and controls the size of the trace table, the type of trace, and whether the trace table should be dumped unconditionally upon termination of the application. |
| TRAP               | When TRAP is set to ON, Language Environment traps routine interrupts and abends, and optionally prints trace information or invokes a user-written condition handling routine.  
  With TRAP set to OFF, the operating system handles all interrupts and abends. You should generally set TRAP to ON, or your runtime results can be unpredictable. |
| USRHDLR            | Specifies the behavior of two user-written condition handlers. The first handler specified will be registered at stack frame 0.  
  The second handler specified will be registered before any other user-written condition handlers, once the handler is enabled by a condition. |
| XUFLOW             | Specifies if an exponent underflow causes a routine interrupt.                                                                                   |

**Determining runtime options in effect**

The runtime options in effect at the time the routine is run can affect routine behavior. Use RPTOPTS(ON) to generate an options report in the Language Environment message file when your routine terminates. The options report lists runtime options, and indicates where they were set. Figure 1 on page 12 shows a
sample options report.
Figure 1. Options report example produced by runtime option RPTOPTS(ON)
Understanding the HEAPZONES and HEAPCHK runtime options

The HEAPZONES and HEAPCHK runtime options are useful for debugging overlay damage problems that occur in the user heap. Though similar in that both options can be used for debugging purposes, the runtime options activate very different behavior in the runtime when specified.

HEAPZONES is a lightweight mechanism that detects heap overlay damage only during the freeing of an element. It looks for damage in the heap check zone of the freed element only.

Selecting a non-quiet output option causes HEAPZONES to display information about the damaged heap element. When messaging is requested, the address of the damaged element along with information specific to the heap check zone are included in the message. Depending on the type of damage, the value of the heap check zone is displayed. The data area of the damaged location is displayed following any issued informational messages. This runtime option can also be used as a mechanism to tolerate heap overlay damage by simply requesting no output (QUIET).

Depending on the size of the heap check zone and the number of allocation requests, the user may notice a significant amount of extra storage being used by the application. Performance may be affected due to the overhead of examining each heap check zone.

HEAPCHK investigates the entire user heap for damage during heap related calls at a frequency based on the specified settings in the option. Because HEAPCHK will traverse the entire user heap, a slow down in application performance will occur. Information about HEAPCHK diagnostic output is discussed in Chapter 3, “Using Language Environment debugging facilities,” on page 39.

When deciding which runtime option is better suited to use with your application, consider the differences between HEAPZONES and HEAPCHK relating to performance, storage usage, and time of damage detection. Although both runtime options affect performance, an application that chooses HEAPCHK will perform slower than an application that chooses HEAPZONES. If storage usage is a concern, HEAPCHK will not consume extra amounts of storage in the manner that HEAPZONES will. Determining when heap damage has occurred may be simpler to accomplish if HEAPCHK is chosen because of the frequency and scope of its analysis.

For more information about the HEAPZONES and HEAPCHK runtime options, see z/OS Language Environment Programming Reference.

Using the CLER CICS transaction to display and set runtime options

The CICS transaction CLER allows you to display all the current Language Environment runtime options for a region, and to modify a subset of these options. For more information about the CICS CLER transaction, see “Displaying and modifying runtime options with the CLER transaction” on page 346.
Controlling storage allocation

The following runtime options control storage allocation:
- STACK
- THREADSTACK
- LIBSTACK
- THREADHEAP
- HEAP
- ANYHEAP
- BELOWHEAP
- STORAGE
- HEAPPOOLS

*z/OS Language Environment Programming Guide* provides useful tips to assist with the tuning process. Appropriate tuning is necessary to avoid performance problems.

To generate a report of the storage a routine (or more specifically, an enclave) used during its run, specify the RPTSTG(ON) runtime option. The storage report, generated during enclave termination provides statistics that can help you understand how space is being consumed as the enclave runs. If storage management tuning is desired, the statistics can help you set the corresponding storage-related runtime options for future runs. The output is written to the Language Environment message file.

Neither the storage report nor the corresponding runtime options include the storage that Language Environment acquires during early initialization, before runtime options processing, and before the start of space management monitoring. In addition, Language Environment does not report alternative Vendor Heap Manager activity.

*Figure 2 on page 15* and *Figure 4 on page 17* are examples of storage reports that are produced when RPTSTG(ON) is specified. The sections that follow these reports describe the contents of the reports.
<table>
<thead>
<tr>
<th>STACK statistics:</th>
<th>THREADSTACK statistics:</th>
<th>LIBSTACK statistics:</th>
<th>THREADHEAP statistics:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial size:</td>
<td>Initial size:</td>
<td>Initial size:</td>
<td>Initial size:</td>
</tr>
<tr>
<td>4096</td>
<td>4096</td>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>Increment size:</td>
<td>Increment size:</td>
<td>Increment size:</td>
<td>Increment size:</td>
</tr>
<tr>
<td>4096</td>
<td>4096</td>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>Maximum used by all concurrent threads:</td>
<td>Maximum used by all concurrent threads:</td>
<td>Maximum used by all concurrent threads:</td>
<td>Maximum used by all concurrent threads:</td>
</tr>
<tr>
<td>7488</td>
<td>3352</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Largest used by any thread:</td>
<td>Largest used by any thread:</td>
<td>Largest used by any thread:</td>
<td>Largest used by any thread:</td>
</tr>
<tr>
<td>7488</td>
<td>3352</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of segments allocated:</td>
<td>Number of segments allocated:</td>
<td>Number of segments allocated:</td>
<td>Number of segments allocated:</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of segments freed:</td>
<td>Number of segments freed:</td>
<td>Number of segments freed:</td>
<td>Number of segments freed:</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2. Storage report produced by runtime option RPTSTG(ON)
HEAP statistics:
Initial size: 49152
Increment size: 16384
Total heap storage used (sugg. initial size): 29112
Successful Get Heap requests: 251
Successful Free Heap requests: 218
Number of segments allocated: 1
Number of segments freed: 0

HEAP24 statistics:
Initial size: 8192
Increment size: 4096
Total heap storage used (sugg. initial size): 0
Successful Get Heap requests: 0
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0

ANYHEAP statistics:
Initial size: 32768
Increment size: 16384
Total heap storage used (sugg. initial size): 104696
Successful Get Heap requests: 28
Successful Free Heap requests: 15
Number of segments allocated: 6
Number of segments freed: 5

BELOWHEAP statistics:
Initial size: 8192
Increment size: 8192
Total heap storage used (sugg. initial size): 0
Successful Get Heap requests: 0
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0

Additional Heap statistics:
Successful Create Heap requests: 1
Successful Discard Heap requests: 1
Total heap storage used: 4912
Successful Get Heap requests: 3
Successful Free Heap requests: 3
Number of segments allocated: 2
Number of segments freed: 2
Largest number of threads concurrently active: 2

End of Storage Report

Figure 3. Storage report produced by runtime option RPTSTG(ON) (continued)

Figure 4 on page 17 shows an example of a storage report that is produced with XPLINK.
Figure 4. Storage report produced by RPTSTG(ON) with XPLINK
HEAP24 statistics:
Initial size: 8192
Increment size: 4096
Total heap storage used (suggested initial size): 0
Successful Get Heap requests: 0
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0

ANYHEAP statistics:
Initial size: 16384
Increment size: 8192
Total heap storage used (suggested initial size): 1139712
Successful Get Heap requests: 487
Successful Free Heap requests: 431
Number of segments allocated: 50
Number of segments freed: 36

BELOWHEAP statistics:
Initial size: 8192
Increment size: 4096
Total heap storage used (suggested initial size): 0
Successful Get Heap requests: 0
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0

Additional Heap statistics:
Successful Create Heap requests: 0
Successful Discard Heap requests: 0
Total heap storage used: 0
Successful Get Heap requests: 0
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0

HEAPPOLLS Statistics:
Pool 1 size: 8 Get Requests: 3
   Successful Get Heap requests: 1-8: 3
Pool 2 size: 32 Get Requests: 268
   Successful Get Heap requests: 9-16: 36
   Successful Get Heap requests: 17-24: 3
   Successful Get Heap requests: 25-32: 229
Pool 3 size: 128 Get Requests: 186
   Successful Get Heap requests: 33-40: 3
   Successful Get Heap requests: 41-48: 8
   Successful Get Heap requests: 49-56: 111
   Successful Get Heap requests: 57-64: 4
   Successful Get Heap requests: 65-72: 2
   Successful Get Heap requests: 73-80: 4
   Successful Get Heap requests: 81-88: 6
   Successful Get Heap requests: 89-96: 2
   Successful Get Heap requests: 97-104: 1
   Successful Get Heap requests: 105-112: 5
   Successful Get Heap requests: 113-120: 31
   Successful Get Heap requests: 121-128: 9

Figure 5. Storage report produced by RPTSTG(ON) with XPLINK (continued)
The statistics for initial and incremental allocations of storage types that have a corresponding runtime option differ from the runtime option settings when their values have been rounded up by the implementation, or when allocations larger than the amounts specified were required during execution. All of the following are rounded up to an integral number of double-words:

- Initial STACK allocations
- Initial allocations of THREADSTACK
- Initial allocations of all types of heap
- Incremental allocations of all types of stack and heap

Figure 6. Storage report produced by RPTSTG(ON) with XPLINK (continued)
The runtime options should be tuned appropriately to avoid performance problems. See z/OS Language Environment Programming Guide for tips on tuning.

**Stack storage statistics**

Language Environment stack storage is managed at the thread level; each thread has its own stack-type resources. Table 10 describes the fields in the storage report that contain various statistics about stack storage.

<table>
<thead>
<tr>
<th>Table 10. Storage report fields that display stack storage statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statistics categories</strong></td>
</tr>
<tr>
<td>• STACK</td>
</tr>
<tr>
<td>• THREADSTACK</td>
</tr>
<tr>
<td>• LIBSTACK</td>
</tr>
<tr>
<td>• IPT STACK</td>
</tr>
<tr>
<td>• XPLINK STACK</td>
</tr>
<tr>
<td>• XPLINK THREADSTACK</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Determining the applicable threads
If the application is not a multithreading or PL/I multitasking application, then the STACK statistics are for the one and only thread that executed, and the THREADSTACK statistics are all zero.

If the application is a multithreading or PL/I multitasking application, and THREADSTACK(ON) was specified, then the STACK statistics are for the initial thread (the IPT), and the THREADSTACK statistics are for the other threads. However, if THREADSTACK(OFF) was specified, then the STACK statistics are for all of the threads, initial and other.

Allocating stack storage
Another type of stack, called the reserve stack, is allocated for each thread and used to handle out-of-storage conditions. Its size is controlled by the 4th subparameter of the STORAGE runtime option, but its usage is neither tracked nor reported in the storage report.

In a single-threaded environment, Language Environment allocates the initial segments for STACK, LIBSTACK and reserve stack using GETMAIN. The LIBSTACK initial segment allocation is deferred until first use, except when STACK(,,BELOW,), is in effect. The reserve stack is allocated with STACK. In a multi-threaded POSIX(ON) environment, allocation of stack storage for the initial processing thread (IPT) is the same as the single-threaded environment. For threads other than the IPT, the initial STACK (or THREADSTACK) segment and reserve stack is allocated from ANYHEAP or BELOWHEAP, according to the STACK (or THREADSTACK) location. The initial LIBSTACK segment allocation is again deferred until first use, except when STACK(,,BELOW,), or THREADSTACK(ON,,BELOW,), is in effect. When a STACK, THREADSTACK, or LIBSTACK overflow occurs on any thread, Language Environment obtains the new segment using GETMAIN. The reserve stack does not tolerate overflow.

Heap storage statistics
Language Environment heap storage, other than what is explicitly defined using THREADHEAP, is managed at the enclave level. Each enclave has its own heap-type resources, which are shared by the threads that execute within the enclave. Heap storage defined using THREADHEAP is controlled at the thread level.

Table 11 on page 22 describes the fields in the storage report that contain various statistics about heap storage. These statistics, in all cases, specify totals for the whole enclave. For THREADHEAP, they indicate the total across all threads in the enclave.
Table 11. Storage report fields that display heap storage statistics

<table>
<thead>
<tr>
<th>Statistics categories</th>
<th>Field contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>THREADHEAP</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Initial size</strong></td>
<td>Default initial allocation, as specified by the corresponding runtime option. For HEAP24, the initial size is the value of the <em>initsz24</em> of the HEAP option.</td>
</tr>
<tr>
<td><strong>Increment size</strong></td>
<td>Minimum incremental allocation, as specified by the corresponding runtime option. For HEAP24, the increment size is the value of the <em>incrsz24</em> of the HEAP option.</td>
</tr>
<tr>
<td><strong>Maximum used by all concurrent threads</strong></td>
<td>Maximum total amount used by all concurrent threads at any one time.</td>
</tr>
<tr>
<td><strong>Largest used by any thread</strong></td>
<td>Largest amount used by any single thread</td>
</tr>
<tr>
<td><strong>Successful Get Heap requests</strong></td>
<td>Number of Get Heap requests.</td>
</tr>
<tr>
<td><strong>Successful Free Heap requests</strong></td>
<td>Number of Free Heap requests.</td>
</tr>
<tr>
<td><strong>Number of segments allocated</strong></td>
<td>Number of incremental segments allocated.</td>
</tr>
<tr>
<td><strong>Number of segments freed</strong></td>
<td>Number of incremental segments individually freed.</td>
</tr>
</tbody>
</table>

• **HEAP**
• **HEAP24**
• **ANYHEAP**
• **BELOWHEAP**

**Initial size**
Default initial allocation, as specified by the corresponding runtime option. For HEAP24, the initial size is the value of the *initsz24* of the HEAP option.

**Increment size**
Minimum incremental allocation, as specified by the corresponding runtime option. For HEAP24, the increment size is the value of the *incrsz24* of the HEAP option.

**Total heap storage used**
Largest total amount used by the enclave at any one time.

**Successful Get Heap requests**
Number of Get Heap requests.

**Successful Free Heap requests**
Number of Free Heap requests. The number of Free Heap requests could be less than the number of Get Heap requests if the pieces of heap storage acquired by individual Get Heap requests were not freed individually, but rather were freed implicitly in the course of enclave termination.

**Number of segments allocated**
Number of incremental segments allocated.

**Number of segments freed**
Number of incremental segments individually freed. The number of incremental segments individually freed could be less than the number allocated if the segments were not freed individually, but rather were freed implicitly in the course of enclave termination.
Table 11. Storage report fields that display heap storage statistics (continued)

<table>
<thead>
<tr>
<th>Statistics categories</th>
<th>Field contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additional heap statistics</td>
<td>Besides the fixed types of heap, additional types of heap can be created, each with its own heap ID. You can create and discard these additional types of heap by using the CEECRHP callable service.</td>
</tr>
<tr>
<td><strong>Successful Create Heap requests</strong></td>
<td>Number of successful Create Heap requests.</td>
</tr>
<tr>
<td><strong>Successful Discard Heap requests</strong></td>
<td>Number of successful Discard Heap requests. The number of Discard Heap requests could be less than the number of Create Heap requests if the special heaps allocated by individual Create Heap requests were not freed individually, but rather were freed implicitly in the course of enclave termination.</td>
</tr>
<tr>
<td><strong>Total heap storage used</strong></td>
<td>Largest total amount used by the enclave at any one time.</td>
</tr>
<tr>
<td><strong>Successful Get Heap requests</strong></td>
<td>Number of Get Heap requests.</td>
</tr>
<tr>
<td><strong>Successful Free Heap requests</strong></td>
<td>Number of Free Heap requests.</td>
</tr>
<tr>
<td><strong>Number of segments allocated</strong></td>
<td>Number of incremental segments allocated.</td>
</tr>
<tr>
<td><strong>Number of segments freed</strong></td>
<td>Number of incremental segments individually freed.</td>
</tr>
</tbody>
</table>

HEAPPOOLS storage statistics

The HEAPPOOLS runtime option (for C/C++ applications only) controls usage of the HEAPPOOLS storage algorithm at the enclave level. The HEAPPOOLS algorithm allows for the definition of one to twelve heap pools, each consisting of a number of storage cells of a specified length. For further details regarding HEAPPOOLS storage statistics in the storage report, see “HEAPPOOLS storage statistics” on page 228.

Modifying condition handling behavior

Setting the condition handling behavior of your routine affects the response that occurs when the routine encounters an error. You can modify condition handling behavior in the following ways:

- Callable services
- Runtime options
- User-written condition handlers
- POSIX functions (used to specifically set signal actions and signal masks)

Language Environment callable services

Table 12 on page 24 lists the callable services that you can use to modify condition handling. For more information about callable services, see z/OS Language Environment Programming Reference. Note that Fortran programs cannot directly call Language Environment callable services. For more information about how to invoke callable services from Fortran, see Language Environment for MVS & VM Fortran Run-Time Migration Guide.
Table 12. Callable services that modify condition handling

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE3ABD</td>
<td>Terminates an enclave using an abend.</td>
</tr>
<tr>
<td>CEE3AB2</td>
<td>Terminate enclave with an abend and reason code.</td>
</tr>
<tr>
<td>CEEMRCE</td>
<td>Moves the resume cursor to an explicit location where resumption is to occur after a condition has been handled.</td>
</tr>
<tr>
<td>CEEMRCR</td>
<td>Moves the resume cursor relative to the current position of the handle cursor.</td>
</tr>
<tr>
<td>CEE3CIB</td>
<td>Returns a pointer to a condition information block (CIB) associated with a given condition token. The CIB contains detailed information about the condition.</td>
</tr>
<tr>
<td>CEE3GRO</td>
<td>Returns the offset of the location within the most current Language Environment-conforming routine where a condition occurred.</td>
</tr>
</tbody>
</table>
| CEE3SPM | Specifies the settings of the routine mask. The routine mask controls:  
|         | • Fixed overflow  
|         | • Decimal overflow  
|         | • Exponent underflow  
|         | • Significance  
|         | You can use CEE3SPM to modify Language Environment hardware conditions. Because such modifications can affect the behavior of your routine, however, you should be careful when doing so. |
| CEE3SRP | Sets a resume point within user application code to resume from a Language Environment user condition handler. |

Language Environment runtime options

Table 13 shows the Language Environment runtime options that can affect your routine’s condition handling behavior.

Table 13. Runtime options that modify condition handling

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description</th>
</tr>
</thead>
</table>
| ABPERC         | Specifies a system- or user-specified abend code that percolates without further action while the Language Environment condition handler is enabled. Normal condition handling activities are performed for everything except the specified abend code. System abends are specified as Shhhh, where hhhh is a hexadecimal system abend code. User abends are specified as Udddd, where dddd is a decimal user abend code. Any other 4-character EBCDIC string, such as NONE, that is not of the form Shhh can also be specified as a user-specified abend code. You can specify only one abend code with this option. This option assumes the use of TRAP(ON). ABPERC is not supported in CICS.  
<p>|                 | Language Environment ignores ABPERC(0Cx). No abend is percolated and Language Environment condition handling semantics are in effect.                                                                      |
| CHECK          | Specifies that checking errors within an application are detected. The Language Environment-conforming languages can define error checking differently.                                                             |</p>
<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTHCONDLMT</td>
<td>Limits the extent to which synchronous conditions can be nested in a user-written condition handler. (Asynchronous signals do not affect DEPTHCONDLMT.) For example, if you specify 5, the initial condition and four nested conditions are processed. If the limit is exceeded, the application terminates with abend code 4091 and reason code 21 (X'15').</td>
</tr>
<tr>
<td>ERRCOUNT</td>
<td>Specifies the number of synchronous conditions of severity 2, 3, and 4 that are tolerated before the enclave terminates abnormally. (Asynchronous signals do not affect ERRCOUNT.) If you specify 0 an unlimited number of conditions is tolerated.</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>Causes attentions recognized by the host operating system to be passed to and recognized by Language Environment after the environment has been initialized.</td>
</tr>
<tr>
<td>TERMTHDACT</td>
<td>Sets the level of information that is produced when a condition of severity 2 or greater remains unhandled within the enclave. The parameter settings for different levels of information include:</td>
</tr>
<tr>
<td></td>
<td>- QUIET for no information</td>
</tr>
<tr>
<td></td>
<td>- MSG for message only</td>
</tr>
<tr>
<td></td>
<td>- TRACE for message and a traceback</td>
</tr>
<tr>
<td></td>
<td>- DUMP for message, traceback, and Language Environment dump</td>
</tr>
<tr>
<td></td>
<td>- UAONLY for message and a system dump of the user address space</td>
</tr>
<tr>
<td></td>
<td>- UATRACE for message, Language Environment dump with traceback information only, and a system dump of the user address space</td>
</tr>
<tr>
<td></td>
<td>- UADUMP for message, traceback, Language Environment dump, and system dump</td>
</tr>
<tr>
<td></td>
<td>- UAIMM for a system dump of the user address space of the original abend or program interrupt prior to the Language Environment condition manager processing the condition.</td>
</tr>
<tr>
<td>TRAP(ON)</td>
<td>Fully enables the Language Environment condition handler. This causes the Language Environment condition handler to intercept error conditions and routine interrupts. During typical operation, you should use TRAP(ON) when running your applications. When TRAP(ON,NOSPIE) is specified, Language Environment handles all program interrupts and abends through an ESTAE. Use this feature when you do not want Language Environment to issue an ESPIE macro.</td>
</tr>
<tr>
<td>TRAP(OFF)</td>
<td>Disables the Language Environment condition handler from handling abends and program checks/interrupts. ESPIE is not issued with TRAP(OFF), it is still possible to invoke the condition handler through the CEESGL callable service and pass conditions to registered user-written condition handlers. Specify TRAP(OFF) when you do not want Language Environment to issue an ESTAE or an ESPIE. However, TRAP(OFF) can cause several unexpected side effects. For more information, see the TRAP runtime option in z/OS Language Environment Programming Reference. When TRAP(OFF), TRAP(OF,ESPIE), or TRAP(OF,NOSPIE) is specified and either a program interrupt or abend occurs, the user exit for termination is ignored.</td>
</tr>
</tbody>
</table>
Table 13. Runtime options that modify condition handling  (continued)

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USRHDLR</td>
<td>Specifies the behavior of two user-written condition handlers. The first handler specified will be registered at stack frame 0. The second handler specified will be registered before any other user-written condition handlers, once the handler is enabled by a condition.</td>
</tr>
<tr>
<td></td>
<td>When you specify USRHDLR(lastname,supername), lastname gets control at stack frame 0. The supername will get control first, before any user-written condition handlers but after supername has gone through the enablement phase, when a condition occurs.</td>
</tr>
<tr>
<td>XUFLOW</td>
<td>Specifies if an exponent underflow causes a routine interrupt.</td>
</tr>
</tbody>
</table>

Customizing condition handlers

User-written condition handlers permit you to customize condition handling for certain conditions. You can register a user-written condition handler for the current stack frame by using the CEEHDLR callable service. You can use the Language Environment USRHDLR runtime option to register a user-written condition handler for stack frame 0. You can also use USRHDLR to register a user-written condition handler before any other user condition handlers.

When the Language Environment condition manager encounters the condition, it requests that the condition handler associated with the current stack frame handle the condition. If the condition is not handled, the Language Environment condition manager percolates the condition to the next (earlier) stack frame, and so forth to earlier stack frames until the condition has been handled. Conditions that remain unhandled after the first (earliest) stack frame has been reached are presented to the Language Environment condition handler. One of the following Language Environment default actions is then taken, depending on the severity of the condition:

- Resume
- Percolate
- Promote
- Fix-up and resume

For more information about user-written condition handlers and the Language Environment condition manager, see z/OS Language Environment Programming Guide.

Invoking the assembler user exit

For debugging purposes, the CEEBXITA assembler user exit can be invoked during:

- Enclave initialization
- Enclave termination
- Process termination

The functions of the CEEBXITA user exit depend on when the user exit is invoked and whether it is application-specific or installation-wide. Application-specific user exits must be linked with the application load module and run only when that application runs. Installation-wide user exits must be linked with the Language Environment initialization/termination library routines and run with all Language Environment library routines. Because an application-specific user exit has priority...
over any installation-wide user exit, you can customize a user exit for a particular application without affecting the user exit for any other applications.

At enclave initialization, the CEEBXITA user exit runs prior to the enclave establishment. Thus you can modify the environment in which your application runs in the following ways:
   • Specify runtime options
   • Allocate data sets/files in the user exit
   • List abend codes to be passed to the operating system
   • Check the values of routine arguments

At enclave termination, the CEEBXITA user exit runs prior to the termination activity. Thus, you can request an abend and perform specified actions based on received return and reason codes. (This does not apply when Language Environment terminates with an abend.)

At process termination, the CEEBXITA user exit runs after the enclave termination activity completes. Thus you can request an abend and deallocate files.

The assembler user exit must have an entry point of CEEBXITA, must be reentrant, and must be capable of running in AMODE(ANY) and RMODE(ANY).

You can use the assembler user exit to establish enclave termination behavior for an enclave ending with an unhandled condition of severity 2 or greater in the following ways:
   • If you do not request an abend in the assembler user exit for the enclave termination call, Language Environment honors the setting of the ABTERMENC option to determine how to end the enclave.
   • If you request an abend in the assembler user exit for the enclave termination call, Language Environment issues an abend to end the enclave.

For more information on the assembler user exit, see z/OS Language Environment Programmming Guide.

Establishing enclave termination behavior for unhandled conditions

To establish enclave termination behavior when an unhandled condition of severity 2 or greater occurs, use one of the following methods:
   • The assembler user exit (see “Invoking the assembler user exit” on page 26 and z/OS Language Environment Programming Guide)
   • POSIX signal default action (see z/OS Language Environment Programming Guide)
   • The ABTERMENC runtime option (discussed below)

The ABTERMENC runtime option sets the enclave termination behavior for an enclave ending with an unhandled condition of severity 2 or greater.

If you specify the IBM-supplied default suboption ABEND, Language Environment issues an abend to end the enclave regardless of the setting of the CEEAUE_ABND flag. Additionally, the assembler user exit can alter the abend code, abend reason code, abend dump attribute, and the abend task/step attribute. For more information on using ABTERMENC, see z/OS Language Environment Programming Reference, and for more information on the assembler user exit, see z/OS Language Environment Programming Guide.
If you specify the RETCODE suboption, Language Environment uses the CEEAU Produkte ABND flag value set by the assembler user exit (which is called for enclave termination) to determine whether or not to issue an abend to end the enclave when an unhandled condition of severity 2 or greater occurs.

### Using messages in your routine

You can create messages and use them in your routine to indicate the status and progress of the routine during run time, and to display variable values. The process of creating messages and using them requires that you create a message source file, and convert the source file into loadable code for use in your routine.

You can use the Language Environment callable service CEEMOUT to direct user-created message output to the Language Environment message file. To direct the message output to another destination, use the Language Environment MSGFILE runtime option to specify the ddname of the file.

When multiple Language Environment environments are running in the same address space and the same MSGFILE ddname is specified, writing contention can occur. To avoid contention, use the MSGFILE suboption ENQ. ENQ tells Language Environment to perform serialization around writes to the MSGFILE ddname specified which eliminates writing contention. Writing contention can also be eliminated by specifying unique MSGFILE ddnames.

Each Language Environment-conforming language also provides ways to display both user-created and runtime messages. (For an explanation of Language Environment runtime messages, see “Interpreting runtime messages” on page 35.)

The following sections discuss how to create messages in each of the HLLs. For a more detailed explanation of how to create messages and use them in C, C/C++, COBOL, Fortran, or PL/I routines, see z/OS Language Environment Programming Guide.

### C/C++

For C/C++ routines, output from the printf function is directed to stdout, which is associated with SYSPRINT. All C/C++ runtime messages and perror() messages are directed to stderr. stderr corresponds to the ddname associated with the Language Environment MSGFILE runtime option. The destination of the printf function output can be changed by using the redirection >>&2 at routine invocation to redirect stdout to the stderr destination. Both streams can be controlled by the MSGFILE runtime option.

### COBOL

For COBOL programs, you can use the DISPLAY statement to display messages. Output from the DISPLAY statement is directed to SYSOUT. SYSOUT is the IBM-supplied default for the Language Environment message file. The OUTDD compiler option can be used to change the destination of the DISPLAY messages.

### Fortran

For Fortran programs, runtime messages, output written to the print unit, and other output (such as output from the SDUMP callable service) are directed to the file specified by the MSGFILE runtime option. If the print unit is different than the
error message unit (PRTUNIT and ERRUNIT runtime options have different values), however, output from the PRINT statement won't be directed to the Language Environment message file.

**PL/I**

Under PL/I, runtime messages are directed to the file specified in the Language Environment MSGFILE runtime option, instead of the PL/I SYSPRINT STREAM PRINT file. User-specified output is still directed to the PL/I SYSPRINT STREAM PRINT file. To direct this output to the Language Environment MSGFILE file, specify the runtime option MSGFILE(SYSPRINT).

---

**Using condition information**

If a condition that might require attention occurs while an application is running, Language Environment builds a condition token. The condition token contains 12 bytes (96 bits) of information about the condition that Language Environment or your routines can use to respond appropriately. Each condition is associated with a single Language Environment runtime message. You can use this condition information in two primary ways:

- To specify the feedback code parameter when calling Language Environment services (see "Using the feedback code parameter").
- To code a symbolic feedback code in a user-written condition handler (see “Using the symbolic feedback code” on page 31).

**Using the feedback code parameter**

The feedback code is an optional parameter of the Language Environment callable services. (For COBOL/370 programs, you must provide the fc parameter in each call to a Language Environment callable service. For C/C++, Enterprise COBOL for z/OS, COBOL for OS/390, COBOL for MVS & VM, and PL/I routines, this parameter is optional. For more information about fc and condition tokens, see [z/OS Language Environment Programming Guide](#).

When you provide the feedback code (fc) parameter, the callable service in which the condition occurs sets the feedback code to a specific value called a condition token.

The condition token does not apply to asynchronous signals. For a discussion of the distinctions between synchronous signals and asynchronous signals with POSIX(ON), see [z/OS Language Environment Programming Guide](#).

When you do not provide the fc parameter, any nonzero condition is signaled and processed by Language Environment condition handling routines. If you have registered a user-written condition handler, Language Environment passes control to the handler, which determines the next action to take. If the condition remains unhandled, Language Environment writes a message to the Language Environment message file. The message is the translation of the condition token into English (or another supported national language).

Language Environment provides callable services that can be used to convert condition tokens to routine variables, messages, or signaled conditions. [Table 14 on page 30](#) lists these callable services and their functions.
Table 14. Callable services that can convert condition tokens to routine variables, messages, or signaled conditions

<table>
<thead>
<tr>
<th>Callable service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEMSG</td>
<td>Transforms the condition token into a message and writes the message to the message file.</td>
</tr>
<tr>
<td>CEEMGET</td>
<td>Transforms the condition token into a message and stores the message in a buffer.</td>
</tr>
<tr>
<td>CEEDCOD</td>
<td>Decodes the condition token; that is, separates it into distinct user-supplied variables. Also, if a language does not support structures, CEEDCOD provides direct access to the token.</td>
</tr>
<tr>
<td>CEESGL</td>
<td>Signals the condition. This passes control to any registered user-written condition handlers. If a user-written condition handler does not exist, or the condition is not handled, Language Environment by default writes the corresponding message to the message file and terminates the routine for severity 2 or higher. For severity 0 and 1, Language Environment continues without writing a message. COBOL, however, issues severity 1 messages before continuing. CEESGL can signal a POSIX condition. For details, see <a href="#">z/OS Language Environment Programming Guide</a>.</td>
</tr>
</tbody>
</table>

There are two types of condition tokens. Case 1 condition tokens contain condition information, including the Language Environment message number. All Language Environment callable services and most application routines use case 1 condition tokens. Case 2 condition tokens contain condition information and a user-specified class and cause code. Application routines, user-written condition handlers, assembler user exits, and some operating systems can use case 2 condition tokens.

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 31</td>
<td>Condition_ID</td>
</tr>
<tr>
<td>32 - 33</td>
<td>Case Number</td>
</tr>
<tr>
<td>34 - 36</td>
<td>Severity Number</td>
</tr>
<tr>
<td>37 - 39</td>
<td>Control Code</td>
</tr>
<tr>
<td>40 - 63</td>
<td>Facility_ID</td>
</tr>
<tr>
<td>64 - 95</td>
<td>ISI</td>
</tr>
</tbody>
</table>

For Case 1 condition tokens, Condition_ID is:
- 0 - 15: Severity Number
- 16 - 31: Message Number

For Case 2 condition tokens, Condition_ID is:
- 0 - 15: Class Code
- 16 - 31: Cause Code

A symbolic feedback code represents the first 8 bytes of a condition token. It contains the Condition_ID, Case Number, Severity Number, Control Code, and Facility_ID, whose bit offsets are indicated.

Figure 7. Language Environment condition token

For example, in the condition token: X'00003032D 59C3C5C5 00000000'
- X'0003' is severity.
- X'032D' is message number 813.
- X'59' are hexadecimal flags for case, severity, and control.
- X'C3C5C5' is the CEE facility ID.
- X'00000000' is the ISI. (In this case, no ISI was provided.)

If a Language Environment traceback or dump is generated while a condition token is being processed or when a condition exists, Language Environment writes the runtime message to the condition section of the traceback or dump. If a condition is detected when a callable service is invoked without a feedback code,
the condition token is passed to the Language Environment condition manager. The condition manager polls active condition handlers for a response. If a condition of severity 0 or 1 remains unhandled, Language Environment resumes without issuing a message. Language Environment does issue messages, however, for COBOL severity 1 conditions. For unhandled conditions of severity 2 or greater, Language Environment issues a message and terminates. For a list of Language Environment runtime messages and corrective information, see z/OS Language Environment Runtime Messages.

If a second condition is raised while Language Environment is attempting to handle a condition, the message CEE0374C CONDITION = <message no.> is displayed using a write-to-operator (WTO). The message number in the CEE0374C message indicates the original condition that was being handled when the second condition was raised. This can happen when a critical error is signaled (for example, when internal control blocks are damaged).

If the output for this error message appears several times in sequence, the conditions appear in order of occurrence. Correcting the earliest condition can cause your application to run successfully.

Using the symbolic feedback code

The symbolic feedback code 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. For more details on symbolic feedback codes, see z/OS Language Environment Programming Guide.
Chapter 2. Classifying errors

This chapter describes errors that commonly occur in Language Environment routines. It also explains how to use runtime messages and abend codes to obtain information about errors in your routine.

Identifying problems in routines

The following sections describe how you can identify errors in Language Environment routines. Included are common error symptoms and solutions.

Language Environment module names

You can identify Language Environment-supplied module elements by any of the following three-character prefixes:

- CEE (Language Environment)
- CEL (Language Environment and C/C++ runtime library)
- EDC (C/C++)
- FOR (Fortran)
- IBM (PL/I)
- IGZ (COBOL)

Module elements or text files with other prefixes are not part of the Language Environment product.

Common errors in routines

These common errors have simple solutions:

- If you do not have enough virtual storage, increase your region size or decrease your storage usage (stack size) by using the storage-related runtime options and callable services. (See “Controlling storage allocation” on page 14 for information about using storage in routines.)
- If you do not have enough disk space, increase your disk allocation.
- If executable files are not available, check your executable library to ensure that they are defined. For example, check your STEPLIB or JOBLIB definitions.

If your error is not caused by any of the items listed above, examine your routine or routines for changes since the last successful run. If there have been changes, review these changes for errors that might be causing the problem. One way to isolate the problem is to branch around or comment out recent changes and rerun the routine. If the run is successful, the error can be narrowed to the scope of the changes.

Duplicate names shared between Fortran routines and C library routines can produce unexpected results. Language Environment provides several cataloged procedures to properly resolve duplicate names. For more information on how to avoid name conflicts, see z/OS Language Environment Programming Guide.

Changes in optimization levels, addressing modes, and input/output file formats can also cause unanticipated problems in your routine.
In most cases, generated condition tokens or runtime messages point to the nature of the error. The runtime messages offer the most efficient corrective action. To help you analyze errors and determine the most useful method to fix the problem, Table 15 lists common error symptoms, possible causes, and programmer responses.

**Table 15. Common error symptoms, possible causes, and programmer responses**

<table>
<thead>
<tr>
<th>Error Symptom</th>
<th>Possible Cause</th>
<th>Programmer Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbered runtime message appears</td>
<td>Condition raised in routine</td>
<td>For any messages you receive, read the Programmer Response. For information about message structure, see “Interpreting runtime messages” on page 35.</td>
</tr>
<tr>
<td>User abend code &lt; 4000</td>
<td>• A non-Language Environment abend occurred.</td>
<td>See the Language Environment abend codes in z/OS Language Environment Runtime Messages. Check for a subsystem-generated abend or a user-specified abend.</td>
</tr>
<tr>
<td></td>
<td>• The assembler user exit requested an abend for an unhandled condition of severity ≥2.</td>
<td></td>
</tr>
<tr>
<td>User abend code ≥ 4000</td>
<td>• Language Environment detected an error and could not proceed.</td>
<td>For any abends you receive, read the appropriate explanation listed in the abend codes section of z/OS Language Environment Runtime Messages.</td>
</tr>
<tr>
<td></td>
<td>• An unhandled software-raised condition occurred and ABTERMENC(ABEND) was in effect.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The assembler user exit requested an abend for an unhandled condition of severity 4.</td>
<td></td>
</tr>
<tr>
<td>System abend with TRAP(OFF)</td>
<td>Cause depends on type of malfunction</td>
<td>Respond appropriately. See the messages and codes book of the operating system.</td>
</tr>
<tr>
<td>System abend with TRAP(ON)</td>
<td>System-detected error</td>
<td>See the messages and codes information of the operating system.</td>
</tr>
<tr>
<td>No response (wait/loop)</td>
<td>Application logic failure</td>
<td>Check routine logic. Ensure ERRCOUNT and DEPTHCONDLMT runtime options are set to a nonzero value.</td>
</tr>
<tr>
<td>Unexpected message (message received was not from most recent service)</td>
<td>Condition caused by something related to current service</td>
<td>Generate a traceback using CEE3DMP.</td>
</tr>
<tr>
<td>Incorrect output</td>
<td>Incorrect file definitions, storage overlay, incorrect routine mask setting, references to uninitialized variables, data input errors, or application routine logic error</td>
<td>Correct the appropriate parameters.</td>
</tr>
<tr>
<td>No output</td>
<td>Incorrect ddname, file definitions, or message file setting</td>
<td>Correct the appropriate parameters.</td>
</tr>
<tr>
<td>Nonzero return code from enclave</td>
<td>Unhandled condition of severity 2, 3, or 4, or the return code was issued by the application routine</td>
<td>Check the Language Environment message file for runtime message.</td>
</tr>
<tr>
<td>Unexpected output</td>
<td>Conflicting library module names</td>
<td>See the name conflict resolution steps outlined in z/OS Language Environment Programming Guide.</td>
</tr>
</tbody>
</table>
Interpreting runtime messages

The first step in debugging your routine is to look up any runtime messages. To find runtime messages, check the message file:

- On z/OS, runtime messages are written by default to ddname SYSOUT. If SYSOUT is not specified, then the messages are written to SYSOUT=*.
- On CICS, the runtime messages are written to the CESE transient data QUEUE.

The default message file ddname can be changed by using the MSGFILE runtime option. For information about displaying runtime messages for Language Environment, COBOL, Fortran, or PL/I routines, see z/OS Language Environment Programming Guide.

Runtime messages provide users with additional information about a condition, and possible solutions for any errors that occurred. They can be issued by Language Environment common routines or language-specific runtime routines and contain a message prefix, message number, severity code, and descriptive text.

In the following example Language Environment message:

```
CEE3206S The system detected a specification exception.
```

- The message prefix is CEE.
- The message number is 3206.
- The severity code is S.
- The message text is The system detected a specification exception.

Language Environment messages can appear even though you made no explicit calls to Language Environment services. C/C++, COBOL, and PL/I runtime library routines commonly use the Language Environment services. This is why you can see Language Environment messages even when the application routine does not directly call common runtime services.

Message prefix

The message prefix indicates the Language Environment component that generated the message. The message prefix is the first three characters of the message number and is also the facility ID in the condition token. See the following table for more information about Language Environment runtime messages.

<table>
<thead>
<tr>
<th>Message Prefix</th>
<th>Language Environment Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE</td>
<td>Common run time</td>
</tr>
<tr>
<td>EDC</td>
<td>C/C++ run time</td>
</tr>
<tr>
<td>FOR</td>
<td>Fortran run time</td>
</tr>
<tr>
<td>IBM</td>
<td>PL/I run time</td>
</tr>
<tr>
<td>IGZ</td>
<td>COBOL run time</td>
</tr>
</tbody>
</table>

The messages for the various components can be found in z/OS Language Environment Runtime Messages and in z/OS MVS Diagnosis: Reference.
**Message number**

The message number is the 4-digit number following the message prefix. Leading zeros are inserted if the message number is less than four digits. It identifies the condition raised and references additional condition and programmer response information.

**Severity code**

The severity code is the letter following the message number and indicates the level of attention called for by the condition. Messages with severity of I are informational messages and do not usually require any corrective action. In general, if more than one runtime message appears, the first noninformational message indicates the problem. For a complete list of severity codes, severity values, condition information, and default actions, see **z/OS Language Environment Programming Guide**.

**Message text**

The message text provides a brief explanation of the condition.

---

**Understanding abend codes**

Under Language Environment, abnormal terminations generate abend codes. There are two types of abend codes: 1) user (Language Environment and user-specified) abends and 2) system abends. User abends follow the format of Udddd, where dddd is a decimal user abend code. System abends follow the format of Shhh, where lhhh is a hexadecimal abend code. Language Environment abend codes are usually in the range of 4000 to 4095. However, some subsystem abend codes can also fall in this range. User-specified abends use the range of 0 to 3999. The following figure shows examples of abend codes.

| User (Language Environment) abend code: U4041 |
| User-specified abend code: U0005 |
| System abend code: S80A |

The Language Environment callable service CEE3ABD terminates your application with an abend. You can set the clean-up parameter value to determine how the abend is processed and how Language Environment handles the raised condition. For more information about CEE3ABD and clean-up, see **z/OS Language Environment Programming Reference**.

You can specify the ABTERMENC runtime option to determine what action is taken when an unhandled condition of severity 2 or greater occurs. For more information on ABTERMENC, see “Establishing enclave termination behavior for unhandled conditions” on page 27, as well as **z/OS Language Environment Programming Reference**.

**User abends**

If you receive a Language Environment abend code, see **z/OS Language Environment Runtime Messages** for a list of abend codes, error descriptions, and programmer responses.
System abends

If you receive a system abend code, look up the code and the corresponding information in the publications for the system you are using.

When a system abend occurs, the operating system can generate a system dump. System dumps are written to ddname SYSMDUMP, SYSABEND, or SYSUDUMP. If the DYNDUMP runtime option is used in combination with the TERMTHDACT runtime option, the system dump can be written without the ddname specified. System dumps show the memory state at the time of the condition. See "Generating a system dump" on page 85 for more information about system dumps.

Using edcmtext to obtain information about errno2 values

Language Environment provides the edcmtext utility (similar to bpxmtext), which allows faster error resolution when an errno2 is encountered in Language Environment. Use the edcmtext utility to display errno2 reason code text. This utility produces a description and action for the errno2 value.

The bpxmtext utility calls edcmtext when the errno2 value is in the range reserved for the C runtime library or edcmtext can be invoked directly with the errno2 value as input.

Format

<table>
<thead>
<tr>
<th>z/OS UNIX environment</th>
<th>TSO/E environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>edcmtext errno2_value</td>
<td>EDCMTEXT errno2_value</td>
</tr>
</tbody>
</table>

Description

The edcmtext utility displays the description and action text for C/C++ runtime library errno2 values, no other values are supported by this command. This command is intended as an aid for problem determination.

The errno2_value is specified as 8 hexadecimal characters.

You can specify one of the following in place of a errno2 value to view a help dialog: -h, help, ?. You can also specify the -U option to display the output in uppercase.

Usage notes

- The errno2_values are also accepted in mixed case and with hex digits prefixed with the "0x".
- The range of values for the XL C/C++ runtime library is 0x'C0000000' through 0x'CFFFFFFF'.
- The utility bpxmtext displays the description and action text for reason codes returned from the kernel, in addition to errno2_values returned from the C/C++ runtime library. You should use bpxmtext when the source of the errno2_values is unknown. For more information, see z/OS UNIX System Services Command Reference.
Message returns

If you specify a `-h, help` or `?` in place of the `errno2_value`, the following message is displayed:

Usage: edcmtext errno2_value

If no text is available for the `errno2_value`, the following message is displayed:

`errno2_value`: No information is currently available for this `errno2_value`.

If the `errno2_value` is not comprised of 1-8 hex digits, the following message is displayed:

Usage: edcmtext errno2_value

If the `errno2_value` is not in the C/C++ runtime library range, the following message is displayed:

Notice: The `errno2_value` is not in the C/C++ runtime library range.

If `edcmtext` is not run in a TSO/E or z/OS UNIX environment, the following message is displayed:

Error: The environment is not TSO/E or z/OS UNIX.

Examples

The command `edcmtext C00B0021` produces data displayed in the following format:

JrEdc1opsEinval01: The mode argument passed to fopen() or freopen() did not begin with `r, w,` or `a.`
Action: Correct the mode argument. The first keyword of the mode argument must be the open mode. Ensure the open mode is specified first and begins with `r, w,` or `a.`
Source: edc1opst.c

Exit Values

0 Successful completion
2 Failure due to an argument that is not 1–8 hex digits
8 Bad Input due to an `errno2_value` out of the C/C++ runtime range.
14 Environment not TSO/E or z/OS UNIX
>20 Contact IBM due to Internal Error
Chapter 3. Using Language Environment debugging facilities

This chapter describes methods of debugging routines in Language Environment. Currently, most problems in Language Environment and member language routines can be determined through the use of a debugging tool or through information provided in the Language Environment dump.

Debug tools

Debug tools are designed to help you detect errors early in your routine. IBM offers Debug Tool, a comprehensive compile, edit, and debug product that is provided with the C/C++ for MVS/ESA, COBOL for OS/390 & VM, COBOL for MVS & VM, PL/I for MVS & VM, and VisualAge for Java compiler products. IBM Debug Tool for z/OS is also available as a standalone product for debugging XL C/C++ applications. For more information, see the [IBM z/OS Debugger](https://developer.ibm.com/mainframe/products/ibm-zos-debugger).

You can use the IBM Debug Tool to examine, monitor, and control how your routines run, and debug your routines interactively or in batch mode. Debug Tool also provides facilities for setting breakpoints and altering the contents and values of variables. Language Environment runtime options can be used with Debug Tool to debug or analyze your routine. See the Debug Tool publications for a detailed explanation of how to invoke and run Debug Tool. For more information, see the [IBM z/OS Debugger](https://developer.ibm.com/mainframe/products/ibm-zos-debugger).


Language Environment dump service, CEE3DMP

The following sections provide information about using the Language Environment dump service, and describe the contents of the Language Environment dump. The Language Environment dump service can be invoked by the following methods:

- CEE3DMP callable service (non-64-bit only)
- TERMTTHDACT runtime option
- HLL-specific functions

Generating a Language Environment dump with CEE3DMP

For non-64-bit, the CEE3DMP callable service generates a dump of the runtime environment for Language Environment and the member language libraries at the point of the CEE3DMP call. You can call CEE3DMP directly from an application routine.

Depending on the CEE3DMP options you specify, the dump can contain information about conditions, tracebacks, variables, control blocks, stack and heap storage, file status and attributes, and language-specific information.

All output from CEE3DMP is written to the default ddname CEEDUMP. CEEDUMP, by default, sends the output to the SDSF output queue. You can direct
the output from the CEEDUMP to a specific sysout class by using the environment variable, _CEE_DMPTARG=SYSOUT(x), where x is the output class.

Under z/OS UNIX, if the application is running in an address-space created as a result of a fork(), spawn(), spawnp(), vfork(), or one of the exec family of functions, then the CEEDUMP is placed in the HFS in one of the following directories in the specified order:
1. The directory found in environment variable _CEE_DMPTARG, if found.
2. The current working directory, if this is not the root directory (/), the directory is writable, and the CEEDUMP path name does not exceed 1024 characters.
3. The directory found in environment variable TMPDIR (an environment variable that indicates the location of a temporary directory if it is not /tmp).
4. The /tmp directory.

The syntax for CEE3DMP is:

```
Syntax

CEE3DMP(--title--,--options--,fc)--
```

**title**
An 80-byte fixed-length character string that contains a title that is printed at the top of each page of the dump.

**options**
A 255-byte fixed-length character string that contains options that describe the type, format, and destination of dump information. The options are declared as a string of keywords that are separated by blanks or commas. Some options also have suboptions that follow the option keyword, and are contained in parentheses. The last option declaration is honored if there is a conflict between it and any preceding options. [Table 16 on page 41](#) lists the CEE3DMP options and related information.

The IBM-supplied default settings for CEE3DMP are:

```
ENCLAVE(ALL) TRACEBACK
THREAD(CURRENT) FILES VARIABLES NOBLOCKS NOSTORAGE
STACKFRAME(ALL) PAGESIZE(60) FNAME(CEEDUMP)
CONDITION ENTRY NOGENOPTS REGSTOR(96)
```

**fc** *(output)*
A 12-byte feedback token code that indicates the result of a call to CEE3DMP. If specified as an argument, feedback information, in the form of a condition token, is returned to the calling routine. If not specified, and the requested operation was not successfully completed, the condition is signaled to the condition manager.

[Table 16 on page 41](#) summarizes the dump options available to CEE3DMP. For more information about the CEE3DMP callable service and dump options, see [z/OS Language Environment Programming Reference](#). For an example of a Language Environment dump, see "Understanding the Language Environment dump" on page 47.
**Table 16. CEE3DMP options**

<table>
<thead>
<tr>
<th>Dump options</th>
<th>Abbreviation</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLAVE(ALL)</td>
<td>ENCL</td>
<td>Dumps all enclaves associated with the current process. (In ILC applications in which a C/C++ routine calls another member language routine, and that routine in turn calls CEE3DMP, traceback information for the C/C++ routine is not provided in the dump.) This is the default setting for ENCLAVE.</td>
</tr>
<tr>
<td>ENCLAVE(CURRENT)</td>
<td>ENCL(CUR)</td>
<td>Dumps the current enclave.</td>
</tr>
<tr>
<td>ENCLAVE(n)</td>
<td>ENCL(n)</td>
<td>Dumps a fixed number of enclaves, indicated by <em>n</em>.</td>
</tr>
<tr>
<td>THREAD(ALL)</td>
<td>THR(ALL)</td>
<td>Dumps all threads in this enclave (including in a PL/I multitasking environment).</td>
</tr>
<tr>
<td>THREAD(CURRENT)</td>
<td>THR(CUR)</td>
<td>Dumps the current thread in this enclave.</td>
</tr>
<tr>
<td>TRACEBACK</td>
<td>TRACE</td>
<td>Includes a traceback of all active routines. The traceback shows transfer of control from calls or exceptions. Calls include PL/I transfers of control from BEGIN-END blocks or ON-units.</td>
</tr>
<tr>
<td>NOTRACEBACK</td>
<td>NOTRACE</td>
<td>Does not include a traceback of all active routines.</td>
</tr>
<tr>
<td>FILES</td>
<td>FILE</td>
<td>Includes attributes of all open files. File control blocks are included when the BLOCKS option is also specified. File buffers are included when the STORAGE option is specified.</td>
</tr>
<tr>
<td>NOFILES</td>
<td>NOFILE</td>
<td>Does not include file attributes.</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>VAR</td>
<td>Includes a symbolic dump of all variables, arguments, and registers.</td>
</tr>
<tr>
<td>NOVARIABLES</td>
<td>NOVAR</td>
<td>Does not include variables, arguments, and registers.</td>
</tr>
<tr>
<td>BLOCKS</td>
<td>BLOCK</td>
<td>Dumps control blocks from Language Environment and member language libraries. Global control blocks, as well as control blocks associated with routines on the call chain, are printed. Control blocks are printed for the routine that called CEE3DMP. The dump proceeds up the call chain for the number of routines that are specified by the STACKFRAME option. Control blocks for files are also dumped if the FILES option was specified. See the FILES option for more information. If the TRACE runtime option is set to ON, the trace table is dumped if BLOCKS is specified. If the Heap Storage Diagnostics report is requested using the HEAPCHK runtime option, the report is displayed when BLOCKS is specified.</td>
</tr>
<tr>
<td>NOBLOCKS</td>
<td>NOBLOCK</td>
<td>Does not include control blocks.</td>
</tr>
<tr>
<td>STORAGE</td>
<td>STOR</td>
<td>Dumps the storage used by the routine. The number of routines dumped is controlled by the STACKFRAME option.</td>
</tr>
</tbody>
</table>
### Table 16. CEE3DMP options  (continued)

<table>
<thead>
<tr>
<th>Dump options</th>
<th>Abbreviation</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOSTORAGE</td>
<td>NOSTOR</td>
<td>Suppresses storage dumps.</td>
</tr>
<tr>
<td>STACKFRAME(ALL)</td>
<td>SF(ALL)</td>
<td>Dumps all stack frames from the call chain. This is the default setting for STACKFRAME.</td>
</tr>
<tr>
<td>STACKFRAME(n)</td>
<td>SF(n)</td>
<td>Dumps a fixed number of stack frames, indicated by ( n ), from the call chain. The specific information dumped for each stack frame depends on the VARIABLES, BLOCK, and STORAGE options declarations. The first stack frame dumped is the caller of CEE3DMP, followed by its caller, and proceeding backward up the call chain.</td>
</tr>
<tr>
<td>PAGESIZE(n)</td>
<td>PAGE(n)</td>
<td>Specifies the number of lines, ( n ), on each page of the dump.</td>
</tr>
<tr>
<td>FNAME(s)</td>
<td>FNAME(s)</td>
<td>Specifies the ddname of the file to which the dump is written.</td>
</tr>
<tr>
<td>CONDITION</td>
<td>COND</td>
<td>Dumps condition information for each condition active on the call chain.</td>
</tr>
<tr>
<td>NOCONDITION</td>
<td>NOCOND</td>
<td>For each condition active on the call chain, does not dump condition information.</td>
</tr>
<tr>
<td>ENTRY</td>
<td>ENT</td>
<td>Includes a description of the program unit that called CEE3DMP and the registers on entry to CEE3DMP.</td>
</tr>
<tr>
<td>NOENTRY</td>
<td>NOENT</td>
<td>Does not include a description of the program unit that called CEE3DMP or registers on entry to CEE3DMP.</td>
</tr>
<tr>
<td>GENOPTS</td>
<td>GENO</td>
<td>Generates a runtime options report in the dump output. This will be the default if an unhandled condition occurs, and a CEEDUMP is generated due to the setting of the TERMTHDACT runtime option setting.</td>
</tr>
<tr>
<td>NOGENOPTS</td>
<td>NOGENO</td>
<td>Does not generate a runtime options report in the dump output. NOGENOPTS is the default for user-called dumps.</td>
</tr>
<tr>
<td>REGST(reg_stor_amount)</td>
<td>REGST(reg_stor_amount)</td>
<td>Controls the amount of storage to be dumped around registers. Default is 96 bytes. Specify REGST(0) if no storage around registers is required.</td>
</tr>
</tbody>
</table>

### Generating a Language Environment dump with TERMTHDACT

The TERMTHDACT runtime option produces a dump during program checks, abnormal terminations, or calls to the CEESGL service. You must use TERMTHDACT(DUMP) in conjunction with TRAP(ON) to generate a Language Environment dump. You can use TERMTHDACT to produce a traceback, Language Environment dump, or user address space dump when a thread ends abnormally because of an unhandled condition of severity 2 or greater. If this is the last thread in the process, the enclave goes away. A thread terminating in a non-POSIX environment is analogous to an enclave terminating because Language Environment Version 1 supports only single threads. For information on enclave termination, see [z/OS Language Environment Programming Guide](http://www.ibm.com).
The TERMTHDACT suboptions QUIET, MSG, TRACE, DUMP, UAONLY, UATRACE, UADUMP, and UAIMM control the level of information available. Table 17 lists the suboptions, the levels of information produced, and the destination of each.

Table 17. TERMTHDACT suboptions, level of information, and destinations

<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of Information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIET</td>
<td>No information</td>
<td>No destination.</td>
</tr>
<tr>
<td>MSG</td>
<td>Message</td>
<td>Terminal or ddname specified in MSGFILE runtime option.</td>
</tr>
<tr>
<td>TRACE</td>
<td>Message and Language Environment dump containing only a traceback</td>
<td>Message goes to terminal or ddname specified in MSGFILE runtime option. Traceback goes to CEEDUMP file.</td>
</tr>
<tr>
<td>DUMP</td>
<td>Message and complete Language Environment dump</td>
<td>Message goes to terminal or ddname specified in MSGFILE runtime option. Language Environment dump goes to CEEDUMP file.</td>
</tr>
<tr>
<td>UAONLY</td>
<td>SYSMDUMP, SYSABEND dump, or SYSUDUMP depending on the DD card used in the JCL in z/OS. In CICS, a transaction dump is created. In non-CICS you will get a system dump of your user address space if the appropriate DD statement is used.</td>
<td>Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified; for CICS the transaction dump goes to DFHDMPA or the DFHDMPB data set.</td>
</tr>
<tr>
<td>UATRACE</td>
<td>Message, Language Environment dump containing only a traceback, and a system dump of the user address space</td>
<td>Message goes to terminal or ddname specified in MSGFILE runtime option. Traceback goes to CEEDUMP file. Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified; for CICS the transaction dump goes to DFHDMPA or the DFHDMPB data set.</td>
</tr>
<tr>
<td>UADUMP</td>
<td>Message, Language Environment dump, and SYSMDUMP, SYSABEND dump, or SYSUDUMP depending on the DD card used in the JCL in z/OS. In CICS, a transaction dump is created.</td>
<td>Message goes to terminal or ddname specified in MSGFILE runtime option. Language Environment dump goes to CEEDUMP file. Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified; for CICS the transaction dump goes to DFHDMPA or the DFHDMPB data set.</td>
</tr>
<tr>
<td>Suboption</td>
<td>Level of Information</td>
<td>Destination</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UAIMM</td>
<td>Language Environment generates a system dump of the original abend/program interrupt of the user address space. In CICS, a transaction dump is created. In non-CICS you will get a system dump of your user address space if the appropriate DD statement is used. After the dump is taken by the operating system, Language Environment condition manager continues processing. Note: Under CICS, UAIMM yields UAONLY behavior. Under non-CICS, TRAP(ON,NOSPIE) must be in effect. When TRAP(ON,SPIE) is in effect, UAIMM yields UAONLY behavior. For software raised conditions or signals, UAIMM behaves the same as UAONLY.</td>
<td>Message goes to terminal or ddname specified in MSGFILE runtime option. User address space dump goes to ddname specified for z/OS; or a CICS transaction dump goes to the DFHDMPA or DFHDMPB data set.</td>
</tr>
</tbody>
</table>

The TRACE and UATTRACE suboptions of TERMTHDACT use these dump options:

- CONDITION
- ENCLAVE(ALL)
- FILES
- FNAME(CEEDUMP)
- GENOPTS
- NOBLOCKS
- NOENTRY
- NOSTORAGE
- STACKFRAME(ALL)
- THREAD(ALL)
- TRACEBACK
- VARIABLES

The DUMP and UADUMP suboptions of TERMTHDACT use these dump options:

- BLOCKS
- CONDITION
- ENCLAVE(ALL)
- FILES
- FNAME(CEEDUMP)
- GENOPTS
- NOENTRY
- STACKFRAME(ALL)
- STORAGE
- THREAD(ALL)
- TRACEBACK
- VARIABLES

Although you can modify CEE3DMP options, you cannot change options for a traceback or dump produced by TERMTHDACT.

**Considerations for setting TERMTHDACT options**

The output of TERMTHDACT may vary depending upon which languages and subsystems are processing the request. This section describes the considerations associated with issuing the TERMTHDACT suboptions. For more information about the TERMTHDACT runtime option, see [z/OS Language Environment Programming Reference](#).
• COBOL Considerations
  The following TERMTHDACT suboptions for COBOL are recommended: UAONLY, UATRACE, and UADUMP. A system dump will always be generated when one of these suboptions is specified.

• PL/I Considerations
  After a normal return from a PL/I ERROR ON-unit, or from a PL/I FINISH ON-unit, Language Environment considers the condition unhandled. If a GOTO is not performed and the resume cursor is not moved, then the thread terminates. The TERMTHDACT setting guides the amount of information that is produced, so the message is not presented twice.

• PL/I MTF Considerations
  TERMTHDACT applies to a task that terminates abnormally due to an unhandled condition of severity 2 or higher that is percolated beyond the initial routine's stack frame. All active subtasks that were created from the incurring task will terminate abnormally, but the enclave will continue to run.

• z/OS UNIX Considerations
  – The TERMTHDACT option applies when a thread terminates abnormally. Abnormal termination of a single thread causes termination of the entire enclave. If an unhandled condition of severity 2 or higher percolates beyond the first routine's stack frame the enclave terminates abnormally.
  – If an enclave terminates due to a POSIX default signal action, then TERMTHDACT applies to conditions that result from software signals, program checks, or abends.
  – If running under a shell and Language Environment generates a system dump, then a storage dump is generated to a file based on the kernel environment variable, _BPXK_MDUMP.

• CICS Considerations
  – TERMTHDACT output is written to a transient data queue named CESE, or to the CICS transaction dump, depending on the setting of the CESE|CICSDDS suboption of the TERMTHDACT runtime option. Table 18 shows the behavior of CESE|CICSDDS when they are used with the other suboptions of TERMTHDACT.
  – Because Language Environment does not own the ESTAE, the suboption UAIMM will be treated as UAONLY.
  – All associated Language Environment dumps will be suppressed if termination processing is the result of an EXEC CICS ABEND with NODUMP.
  – Program checks and other abends will cause CICS to produce a CICS transaction dump.

Table 18. Condition handling of 0Cx abends

<table>
<thead>
<tr>
<th>Options</th>
<th>TERMTHDACT(X,CESE,)</th>
<th>TERMTHDACT(X,CICSDDS,)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIET</td>
<td>No output.</td>
<td>No output.</td>
</tr>
<tr>
<td>MSG</td>
<td>Message written to CESE queue or MSGFILE.</td>
<td>Message written to CESE queue or MSGFILE.</td>
</tr>
</tbody>
</table>
Table 18. Condition handling of 0Cx abends (continued)

<table>
<thead>
<tr>
<th>Options</th>
<th>TERMTHDACT(X,CESE,)</th>
<th>TERMTHDACT(X,CICSDDS,)</th>
</tr>
</thead>
</table>
| TRACE   | The traceback is written to the CESE queue, followed by U4038 abend with nodump option. | • Language Environment will write traceback, variables, COBOL working storage, C writeable static. The member handlers will be invoked to provide the desired output to the new transaction server queue (which CICS will read and write to CICS transaction dump later).  
• U4039 abend to force CICS transaction dump followed by U4038 abend with nodump option.  
• Message to CESE or MSGFILE. |
| DUMP    | CEEDUMP to CESE queue followed by U4038 abend with nodump option. | • CEEDUMP to new transaction server queue which CICS will read and write to CICS transaction dump later.  
• U4039 abend to force CICS transaction dump followed by U4038 abend with nodump option.  
• Message to CESE or MSGFILE. |
| UATRACE | U4039 abend with traceback to CESE queue followed by U4038 abend with nodump option. | • Language Environment will write traceback, variables, COBOL working storage, C writeable statics. The member handlers will be invoked to provide the desired output to the new transaction server queue (which CICS will read and write to CICS transaction dump later).  
• U4039 abend to force CICS transaction dump followed by U4038 abend with nodump option.  
• Message to CESE or MSGFILE. |
| UADUMP  | U4039 abend with CEEDUMP to CESE queue followed by U4038 abend with nodump option. | • CEEDUMP to new transaction server queue which CICS will read and write to CICS transaction dump later.  
• U4039 abend to force CICS transaction dump followed by U4038 abend with nodump option.  
• Message to CESE or MSGFILE. |
| UAONLY  | U4039 abend followed by U4038 abend with nodump option. | • U4039 abend followed by U4038 abend with nodump option.  
• No CEEDUMP information is generated.  
• Same as CESE. |
| UAIMM   | U4039 abend followed by U4038 abend with nodump option. | • U4039 abend followed by U4038 abend with nodump option.  
• No CEEDUMP information is generated.  
• Same as CESE. |

Generating a Language Environment dump with language-specific functions

In addition to the CEE3DMP callable service and the TERMTHDACT runtime option, you can use language-specific routines such as C functions, the Fortran SDUMP service, and the PL/I PLIDUMP service to generate a dump.

C/C++ routines can use the functions cdump(), csnap(), and ctrace() to produce a Language Environment dump. All three functions call the CEE3DMP callable service, and each function includes an options string consisting of different CEE3DMP options that you can use to control the information contained in the dump. For more information on these functions, see “Generating a Language Environment dump of a C/C++ routine” on page 190.
Fortran programs can call SDUMP, DUMP/PDUMP, or CDUMP/CPDUMP to generate a Language Environment dump. CEE3DMP cannot be called directly from a Fortran program. For more information on these functions, see “Generating a Language Environment dump of a Fortran routine” on page 261.

PL/I routines can call PLIDUMP instead of CEE3DMP to produce a dump. PLIDUMP includes options that you can specify to obtain a variety of information in the dump. For a detailed explanation about PLIDUMP, see “Generating a Language Environment dump of a PL/I for MVS & VM routine” on page 282.

Understanding the Language Environment dump

The Language Environment dump service generates output of data and storage from the Language Environment runtime environment on an enclave basis. This output contains the information needed to debug most basic routine errors.

This sample illustrates a dump for enclave main. The example assumes full use of the CEE3DMP dump options. Ellipses are used to summarize some sections of the dump and information regarding unhandled conditions may not be present at all. Sections of the dump are numbered to correspond with the descriptions given in “Sections of the Language Environment dump” on page 59.

The CEE3DMP was generated by the C program CELSAMP shown in Figure 8 on page 48. CELSAMP uses the DLL CELDLL shown in Figure 11 on page 51.
#pragma options(SERVICE("1.1.d"),NOOPT,TEST(SYM))
#pragma runopts(TERMTHDACT(UADUMP),POSIX(ON),DYNDUMP(DYNAMIC,))
#pragma runopts(TRACE(ON,IM,NO DUMP,LE=1),HEAPCHK(ON,1,0,10,10))
#pragma runopts(RPTSTG(ON),HEAPPOOLS(ON))
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <dl1.h>
#include <signal.h>
#include <leawi.h>
#include <ceeedcct.h>

pthread_mutex_t mut;
pthread_t thread[2];
int * threads Joined = 0;
char * t1 = "Thread 1";
char * t2 = "Thread 2";

/* thread cleanup: condition handler to clean up threads */
void thread_cleanup(_FEEDBACK *cond, _INT4 *input_token,
                   _INT4 *result, _FEEDBACK *new_cond) {
    *result = percolate;
    printf(">>> Thread Cleanup: Percolating condition\n");
}

/* values for handling the conditions */
#define percolate  20
printf(">>> Thread CleanUp: Msg # is %d\n",cond->tok_msgno);
if (!threads Joined) {
    printf(">>> Thread CleanUp: Locking mutex\n");
    pthread_mutex_lock(&mut);
    printf(">>> Thread CleanUp: Joining threads\n");
    if (pthread_join(thread[0],NULL) == -1 )
        perror("Join of Thread #1 failed");
    if (pthread_join(thread[1],NULL) == -1 )
        perror("Join of Thread #2 failed");
    threads Joined = 1;
}

void *thread_func(void *parm) {
    printf(">>> Thread func: Locking mutex\n",parm);
    pthread_mutex_lock(&mut);
    printf(">>> Thread func: Exiting\n",parm);
    pthread_exit(NULL);
}

Figure 8. The C program CELSAMP
main()
{
dllhandle = handle;
int i = 0;
FILE* fp1;
FILE* fp2;
_FEEDBACK fc;
_INT4 token;
_ENTRY pgmptr;

printf("Init MUTEX...
");
if (pthread_mutex_init(&mut, NULL) == -1) {
    perror("Init of mut failed");
    exit(101);
}

printf("Lock Mutex Lock...
");
if (pthread_mutex_lock(&mut) == -1) {
    perror("Lock of mut failed");
    exit(102);
}

printf("Create 1st thread...
");
if (pthread_create(&thread[0], NULL, thread_func, (void *)t1) == -1) {
    perror("Could not create thread #1");
    exit(103);
}

printf("Create 2nd thread...
");
if (pthread_create(&thread[1], NULL, thread_func, (void *)t2) == -1) {
    perror("Could not create thread #2");
    exit(104);
}

printf("Register thread cleanup condition handler...
");
pgmptr.address = (_POINTER)thread_cleanup;
pgmptr.nesting = NULL;
token = 1;
CEEHDLR (&pgmptr, &token, &fc);
if (!_FBCHECK (fc, CEE000) ) {
    printf("CEEHDLR failed with message number %d\n",fc.tok_msgno);
    exit(105);
}

printf("Load DLL...
");
handle = dllload("CELDLL");
if (handle == NULL) {
    perror("Could not load DLL CELDLL");
    exit(106);
}

printf("Query DLL with incorrect function name...
");
pgmptr.address = (_POINTER)dllqueryfn(handle,"name_not_in_dll");
if (pgmptr.address == NULL) {
    perror("Found incorrect function name in DLL");
    exit(111);
}

printf("Query DLL...
");
pgmptr.address = (_POINTER)dllqueryfn(handle,"dump_n_perc");
if (pgmptr.address == NULL) {
    perror("Could not find dump_n_perc");
    exit(107);
}
printf("Register condition handler...
");
pgmptr.nesting = NULL;
token = 2;
CEEHDLR (&pgmptr, &token, &fc);
if (_FBCCHECK ( fc, CEE000 ) != 0 ) {
    printf("CEEHDLR failed with message number %d\n", 
        fc.tok_msgno);
    exit(108);
}

printf("Write to some files...
");
fp1 = fopen("myfile.data", "w");
if (fp1) {
    perror("Could not open myfile.data for write");
    exit(109);
}
    fprintf(fp1, "record 1\n");
    fprintf(fp1, "record 2\n");
    fprintf(fp1, "record 3\n");

    fp2 = fopen("memory.data", "wb,type=memory");
    if (fp2) {
        perror("Could not open memory.data for write");
        exit(112);
    }

    fprintf(fp2, "some data");
    fprintf(fp2, "some more data");
    fprintf(fp2, "even more data");

    printf("Divide by zero...
");
i = 1/i;
    printf("Error -- Should not get here\n");
    exit(110);
}

Figure 10. The C program CELSAMP (continued)
For easy reference, the sections of the following dump are numbered to correspond with the descriptions in “Sections of the Language Environment dump” on page 59.

Figure 11. The C DLL CELDLL

For easy reference, the sections of the following dump are numbered to correspond with the descriptions in “Sections of the Language Environment dump” on page 59.
Information

VR28.... 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
VR26.... 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
VR24.... 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
VR20.... 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
VR14.... 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
VR12.... 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
VR8...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Storage around GPR1 (265ECF8)
-0020 25EF08 25EF0A 265ECF8 265ECF0 265ED00 265ED09 265ED08 265ED0A
+0000 265EDE8 GPR20 GPR19 GPR18 GPR17 GPR16 GPR15 GPR14 GPR13
+0020 265ED00 265ED0A 265ED04 265ED09 265ED14 265ED19 265ED24 265ED29 265ED34
+0040 265ED39 265ED44 265ED49 265ED54 265ED59 265ED64 265ED69 265ED74 265ED79
+0060 265ED84 265ED89 265ED94 265ED99 265EDA4 265EDAA 265EDB4 265EDB9 265EDC4
+0080 265EDC9 265EDDF 265EE04 265EE09 265EE14 265EE19 265EE24 265EE29 265EE34
+0100 265EE39 265EE44 265EE49 265EE54 265EE59 265EE64 265EE69 265EE74 265EE79
+0120 265EE84 265EE89 265EE94 265EE99 265EEA4 265EEAA 265EED4 265EED9 265EEF4
+0140 265EEF9 265EF04 265EF09 265EF14 265EF19 265EF24 265EF29 265EF34 265EF39
+0160 265EF44 265EF49 265EF54 265EF59 265EF64 265EF69 265EF74 265EF79 265EF84
+0180 265EF89 265EF94 265EF99 265EEA4 265EEA9 265EEB4 265EEB9 265EEC4 265EEC9
+0200 265EEC9 265EEC0 265EEC1 265EEC2 265EEC3 265EEC4 265EEC5 265EEC6 265EEC7
+0220 265EEC8 265EEC9 265EECA 265EECB 265EECC 265EECD 265EECE 265EECF 265EECG

Storage around GPR2 (265EDE8)
-0020 25EF0A 25EF0C 25EF0E 25EF10 25EF12 25EF14 25EF16 25EF18 25EF1A
+0140 25EF9A 25EF9C 25EF9E 25EF00 25EF02 25EF04 25EF06 25EF08 25EF0A
+0160 25EF0C 25EF0E 25EF10 25EF12 25EF14 25EF16 25EF18 25EF1A 25EF1C

Storage around GPR4 (265EDEA)
+0000 265EDE8 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA 265EDE2
+0020 265EDE4 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8
+0040 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4
+0060 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA
+0080 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4
+0100 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA
+0120 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4
+0140 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA
+0160 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4
+0180 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA
+0200 265EDEA 265EDE2 265EDE4 265EDE6 265EDE8 265EDEA 265EDE2 265EDE4

Traceback:
- dump_n_perc_return

52 z/OS Language Environment Debugging Guide
Condition Information for Active Routines

Condition Information for /*POSIX.CRTL.C(CELSAMP)* (DSA address 265E920B)

CIB Address: 265E9A0B

Current Condition: CEE3209S The system detected a fixed-point divide exception (System Completion Code-OC9).

Location:
Program Unit: /*POSIX.CRTL.C(CELSAMP)*
Entry: main Statement: 150 Offset: 0000009B

Machine State:
ILC...... 0002 Interruption Code...... 0009
PSM...... 07B02400 A560A0C

Parameters:
Storage:

+0000 00000000 Inaccessible storage.
+0000 00002000 Inaccessible storage.
+0040 00000040 Inaccessible storage.

Local Variables:
title[6..6] unsigned char 'S' 'A' 'M' 'n' 'p' 'l' 'e'...
title[9..13] unsigned char 'd' 'g' 'm' 'p' 'r' 't' 'p'...
title[21..27] unsigned char 'b' 'y' 'c' 'g' 'l' 't' 'l'...
title[28..34] unsigned char 'i' 'g' 'm' 'p' 'r' 't' 'p'...
title[35..41] unsigned char 'j' 'd' 'M' 'p' 'r' 't' 'l'...
title[42..48] unsigned char 'i' 'g' 'm' 'p' 'r' 't' 'l'...

title[49..55] to title[70..76] elements same as above.
title[77..79] unsigned char 'T' 'A' 'R' 'E' 'A' 'D' 'f'...

Options[0..6]

options[0..6] unsigned char 't' 'A' 'R' 'E' 'A' 'D' 'F'

Options[7..13]

options[7..13] unsigned char 'A' 'L' 'L' 'L' 'L' 'L' 'L'

Options[14..20]

options[14..20] unsigned char 'd' 'c' 'k' 'I' 'S' 'E' 'I'

Options[21..27]

options[21..27] unsigned char 'd' 'c' 'k' 'i' 'S' 'E' 'I'

Options[28..34]

options[28..34] unsigned char 'E' 'M' 'o' 'P' 't' 's' 'm'

Options[35..41]

options[35..41] unsigned char 'E' 'M' 'o' 'P' 't' 's' 'm'

Options[42..48]

Options[42..48] unsigned char 'E' 'M' 'o' 'P' 't' 's' 'm'

Options[252..254]

Options[252..254] unsigned char 'E' 'M' 'o' 'P' 't' 's' 'm'

fc struct _FEEDBACK

fc struct _FEEDBACK

tok_dev signed short int -1310B

tok_msgno signed short int -1310B

tok_case unsigned:2 3

tok_case unsigned:2 3

tok_sever unsigned:3 1

tok_ctrl unsigned:3 4

tok_pic[0..2] unsigned char 'kCC' 'kCC' 'kCC'

tok_pic int -858993460

__func__[0..6] static unsigned char 'g' 'w' 'm' 'p' 'r' 'o'

__func__[7..11] unsigned char 'p' 'e' 'r' 'c' 'd' 'e'

main (DSA address 265E920B):

UPSTACK DSA

Saved Registers:

GPR0.... 00000000_265E9208
GPR1..... 00000000_265E9208
GPR2..... 00000000_265E9208
GPR3..... 00000000_265E9208
GPR4..... 00000000_265E9208
GPR5..... 00000000_265E9208
GPR6..... 00000000_265E9208
GPR7..... 00000000_265E9208
GPR8..... 00000000_265E9208
GPR9..... 00000000_265E9208
GPR10.... 00000000_265E9208
GPR11.... 00000000_265E9208
GPR12.... 00000000_265E9208
GPR13.... 00000000_265E9208
GPR14.... 00000000_265E9208
GPR15.... 00000000_265E9208

Chapter 3. Using Language Environment debugging facilities
[7] Backtrace:

```
Registers
Thread_func
CEEOPML2

Fully Qualified Names
```

[8] Parameters, Registers, and Variables for Active Routines:

```
CEEOPML2 (DS address 2693090)
```

[9] Saved Registers:

```
GPR0:...... 00000000 00000001 00000005 00000000 00000000 00000000 00000000
GPR1:...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR2:...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR3:...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR4:...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR5:...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR6:...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR7:...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR8:...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR9:...... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR10:..... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR11:..... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR12:..... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR13:..... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR14:..... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR15:..... 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Chapter 3. Using Language Environment debugging facilities 55
UPSTACK DSA
Parameters:
parm  void  * Ox2SE0ED0
Saved Registers:
GPR0..... 0000001 GPR1..... 26935070 GPR2..... 2671C43C GPR3..... 26935078
GPR4..... 26EC288 GPR5..... 0000000 GPR6..... 26561A0 GPR7..... 2660A930.

[10] Control Blocks for Active Routines:
DSA for CEEMPUL: 26938090
+000000  FLAGS.... 0000 member.... CCC C BKC.... 269730E0 FWC..... 2693B210 R14..... ASF5EF76
+000100  R15..... ASF60090 R00..... 25F5F204 R1..... 2693B114 R2..... 2671C43C R3..... 26935078
+000200  R4..... 00000000 R5..... 2660A954 R6..... 26561A0 R7..... 2660A930 R8..... 25E00ED0.

[11] Storage for Active Routines:
DSA frame: 269370E0
+000000  269370E0 10CCCCC 26938090 A6411BDE ASF5E148 25F5F204 25E0F158 26561A0
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .....
HeapCHK
Heap CHK
Header:
HEAPCHK
Table:
HEAPCHK
2660AB50

Most
2660D590

Address.................265E6148

Addr
Information:
Storage
------------
Addr
recent
265E6248
2699D048

Element
Option
265E6288
265E6248
2699D048
2699D028
269E0048
2699C020
265E5110
265E50D0
2671CBA4
2671CB84
2699A000
2625860000000000

Trace
Length
40404040
94818995
Member
Time
------------------------------------------------------------------------
Trace
Entry
is
in
displacement:
---(085)

Entry

Heap Storage Diagnostics
Stg Addr ID  Length  Entry  E Addr  E Offset  Load Mod
269F020  269F09FC  00000050

269F070  269F09FC  00000028

2660938  2660918  00001030  2668190  2660468  2660918  00000828  2658258

WSA address.............2656E148

Language Environment Trace Table:
Most recent trace entry is at displacement: 002980

Displacement Trace Entry in Hexadecimal
Trace Entry in EBCDIC
------------------------------------------------------------------------
+00000  Time 15.25.14.S7201  Date 2010.04.07  Thread ID... 2625860000000000
+000010  Member ID:... 00000  Entry Type:... 00000001
+000018  94818995  40400404  40400404  40400404  40400404  40400404  40400404  40400404  40400404
+000058  40400404  40400404  40400404  40400404  40400404  40400404  40400404  40400404
+000078  40400404  40400404  40400404  40400404  40400404  40400404  40400404  40400404  40400404

Chapter 3. Using Language Environment debugging facilities 57
[15] Inclave Storage:

Initial (User) Heap:

LE/370 Anywhere Heap:

LE/370 Anywhere Heap:

Additional Heap, heapid = 26909F0:

[16] File Status and Attributes:

[17] Runtime Options Report:

LAST WHERE SET OPTION

IBM-supplied default ABIPERC(NONE)
IBM-supplied default ABTERMENC(AEND)
IBM-supplied default NOXIEBLD
IBM-supplied default ALL3ON
IBM-supplied default ANYHEAP(16384,8192,ANYWHERE,FREE)
IBM-supplied default NOAUTOTASK
IBM-supplied default BELOWHEAP(B192,4096,FREE)
IBM-supplied default CBLOPTS(ON)
IBM-supplied default CBPSPHOLD(ON)
IBM-supplied default CBLQDA(OFF)
DC:CEEDOPTS CEEDEUMP(0,SYSTOUT=*,FREE=END,SPIN=UNALLOC)
IBM-supplied default CHECK(ON)
IBM-supplied default COUNTRY(US)
IBM-supplied default NODEBUG
IBM-supplied default DEPTHCONDMP(10)
DC:CEEDOPTS DYNDUMP(POSEX,DEBUG,HLE7770,DYNAMIC,)
IBM-supplied default ENVAR(""")
IBM-supplied default ERRCOUNT(0)
IBM-supplied default ERRUNIT(6)
Sections of the Language Environment dump

The sections of the dump listed here appear independently of the Language Environment-conforming languages used. Each conforming language adds language-specific storage and file information to the dump. For a detailed explanation of language-specific dump output:

- For C/C++ routines, see “Finding C/C++ Information in a Language Environment dump” on page 199.
- For COBOL routines, see “Finding COBOL Information in a Dump” on page 244.
- For Fortran routines, see “Finding Fortran Information in a Language Environment Dump” on page 266.
Table 19. Contents of the Language Environment dump

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Page Heading</td>
<td>The page heading section appears on the top of each page of the dump and contains the following information:</td>
</tr>
<tr>
<td></td>
<td>• CEE3DMP identifier</td>
</tr>
<tr>
<td></td>
<td>• Title: For dumps generated as a result of an unhandled condition, the title is “Condition processing resulted in the Unhandled condition.”</td>
</tr>
<tr>
<td></td>
<td>• Product abbreviation of Language Environment</td>
</tr>
<tr>
<td></td>
<td>• Version number</td>
</tr>
<tr>
<td></td>
<td>• Release number</td>
</tr>
<tr>
<td></td>
<td>• Date</td>
</tr>
<tr>
<td></td>
<td>• Time</td>
</tr>
<tr>
<td></td>
<td>• Page number</td>
</tr>
<tr>
<td></td>
<td>The contents of the second line of the page heading vary depending on the environment in which the CEEDUMP is issued.</td>
</tr>
<tr>
<td></td>
<td>For CEEDUMPs produced under a batch environment, the following items are displayed:</td>
</tr>
<tr>
<td></td>
<td>• ASID: Describes the address space ID.</td>
</tr>
<tr>
<td></td>
<td>• Job ID: Describes the JES Job ID.</td>
</tr>
<tr>
<td></td>
<td>• Job name: Describes the job name.</td>
</tr>
<tr>
<td></td>
<td>• Step name: Describes the job’s step name in which the CEEDUMP was produced.</td>
</tr>
<tr>
<td></td>
<td>• UserID: Describes the TSO userid who issued the job.</td>
</tr>
<tr>
<td></td>
<td>For jobs running with POSIX(ON), the following additional items are displayed:</td>
</tr>
<tr>
<td></td>
<td>• PID: Displays the associated process ID.</td>
</tr>
<tr>
<td></td>
<td>• Parent PID: Displays the associated parent PID.</td>
</tr>
<tr>
<td></td>
<td>For CEEDUMPs produced under the z/OS UNIX shell, the following items are displayed:</td>
</tr>
<tr>
<td></td>
<td>• ASID: Describes the address space ID.</td>
</tr>
<tr>
<td></td>
<td>• PID: Displays the associated process ID.</td>
</tr>
<tr>
<td></td>
<td>• Parent PID: Displays the associated parent PID.</td>
</tr>
<tr>
<td></td>
<td>• User name: Contains the user ID associated to the CEEDUMP.</td>
</tr>
<tr>
<td></td>
<td>For CEEDUMPs produced under CICS, the following items are displayed:</td>
</tr>
<tr>
<td></td>
<td>• Transaction ID and task number.</td>
</tr>
<tr>
<td>[2] CEE3845I CEEDUMP Processing started.</td>
<td>Identifies the start of the Language Environment dump processing. Similarly, message CEE3846I identifies the end of the dump processing. Message number CEE3845I can be used to locate the start of the next CEEDUMP report when scanning forward in a data set that contains several CEEDUMP reports.</td>
</tr>
<tr>
<td>[3] Caller Program Unit and Offset</td>
<td>Identifies the routine name and offset in the calling routine of the call to the dump service.</td>
</tr>
</tbody>
</table>
Table 19. Contents of the Language Environment dump (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4] Registers on Entry to CEE3DMP</td>
<td>Shows data at the time of the call to the dump service.</td>
</tr>
<tr>
<td></td>
<td>• Program mask: The program mask contains the bits for the fixed-point overflow mask, decimal overflow mask, exponent underflow mask, and significance mask.</td>
</tr>
<tr>
<td></td>
<td>• General purpose registers (GPRs) 0–15: On entry to CEE3DMP, the GPRs contain:</td>
</tr>
<tr>
<td></td>
<td>GPR 0 Working register</td>
</tr>
<tr>
<td></td>
<td>GPR 1 Pointer to the argument list</td>
</tr>
<tr>
<td></td>
<td>GPR 2–11 Working registers</td>
</tr>
<tr>
<td></td>
<td>GPR 12 Address of CAA</td>
</tr>
<tr>
<td></td>
<td>GPR 13 Pointer to caller’s stack frame</td>
</tr>
<tr>
<td></td>
<td>GPR 14 Address of next instruction to run if the ALL31 runtime option is set to ON</td>
</tr>
<tr>
<td></td>
<td>GPR 15 Entry point of CEE3DMP</td>
</tr>
<tr>
<td></td>
<td>• Floating point registers (FPRs) 0 through 15</td>
</tr>
<tr>
<td></td>
<td>• Vector registers (VRs) 0 through 31.</td>
</tr>
<tr>
<td></td>
<td>• Storage pointed to by General Purpose Registers. Treating the contents of each register as an address, 32 bytes before and 64 bytes after the address are shown.</td>
</tr>
</tbody>
</table>

[5] - [17] Enclave Information. These sections show information that is specific to an enclave. When multiple enclaves are dumped, these sections will appear for each enclave.

If multiple CEEPIPI main-DP environments exist, the dump service generates data and storage information for the most current Main-DP environment, followed by the previous (parent) Main-DP environments in a last-in-first-out (LIFO) order. Sections [5] - [17] will appear for each enclave in the most current Main-DP environment, and sections [5]-[7] will appear for enclaves in the previous (parent) Main-DP environments. When multiple nested Main-DP environments are present in the dump output, a line displaying the CEEPIPI token value for each dumped Main-DP environment will appear before the output for that environment.

[5] Enclave Identifier | Names the enclave for which information in the dump is provided. If multiple enclaves exist, the dump service generates data and storage information for the most current enclave, followed by previous enclaves in a last-in-first-out (LIFO) order. For more information about dumps for multiple enclaves, see “Multiple enclave dumps” on page 83.

[6] - [12] Thread Information. These sections show information that is specific to a thread. When multiple threads are dumped, these sections will appear for each thread.

[6] Information for thread | Shows the system identifier for the thread. Each thread has a unique identifier.
<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7] Traceback</td>
<td>In a multithread case, the traceback reflects only the current thread. For all active routines, the traceback section shows routine information in three parts. The first part contains:</td>
</tr>
<tr>
<td></td>
<td>• DSA number: A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback.</td>
</tr>
<tr>
<td></td>
<td>• Entry: For COBOL, Fortran, PL/I, and Enterprise PL/I for z/OS routines, this is the entry point name. For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, the string '** NoName **' will appear.</td>
</tr>
<tr>
<td></td>
<td>• Entry point offset</td>
</tr>
<tr>
<td></td>
<td>• Statement number: Refers to the line number in the source code (program unit) in which a call was made or an exception took place (see Status column). The statement number appears only if your routine was compiled with the options required to generate statement numbers.</td>
</tr>
<tr>
<td></td>
<td>• Load module: The load module name displayed can be a partitioned data set member or an UNIX executable file. The load module name is also displayed in the third part of the traceback.</td>
</tr>
<tr>
<td></td>
<td>• Program unit: For COBOL programs, program unit is the PROGRAM-ID name. For C, Fortran, and PL/I routines, program unit is the compile unit name. For Language Environment-conforming assemblers, program unit is either the EPNAME = value on the CEEPMA macro, or a fully qualified path name.</td>
</tr>
<tr>
<td></td>
<td>If the program unit name is available to Language Environment (for example, for C/C++, the routine was compiled with TEST(SYM)), the program unit name will appear under this column, according to the following rules:</td>
</tr>
<tr>
<td></td>
<td>– If your compiled routine is in a partitioned data set, only the member will be output.</td>
</tr>
<tr>
<td></td>
<td>– If your compiled routine is in a sequential data set, only the last qualifier will be shown.</td>
</tr>
<tr>
<td></td>
<td>– If your compiled routine is in an UNIX filename, only what fits of the filename will be displayed in a line.</td>
</tr>
<tr>
<td></td>
<td>• Service level: The latest service level applied to the compile unit (for example, for IBM products, it would be the PTF number).</td>
</tr>
<tr>
<td></td>
<td>– If the service level string is equal or less than 7 bytes, all of the string will be output.</td>
</tr>
<tr>
<td></td>
<td>– If the service level string is longer than 7 bytes, the Service column will only show the first 7 bytes of the service string, and the full service string will be shown in section of Full Service Level with max length of 64 bytes.</td>
</tr>
<tr>
<td></td>
<td>• Status: Routine status can be ‘call’ or ‘exception’.</td>
</tr>
</tbody>
</table>
Table 19. Contents of the Language Environment dump (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7] Traceback (continued)</td>
<td>The second part contains:</td>
</tr>
<tr>
<td></td>
<td>• DSA number: A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback.</td>
</tr>
<tr>
<td></td>
<td>• Stack frame (DSA) address</td>
</tr>
<tr>
<td></td>
<td>• Entry point address</td>
</tr>
<tr>
<td></td>
<td>• Program unit address</td>
</tr>
<tr>
<td></td>
<td>• Program unit offset: The offset of the last instruction to run in the routine. If the offset is a negative number, zero, or a very large positive number, the routine associated with the offset probably did not allocate a save area or could have been called using SVC-assisted linkage. Adding the program unit address to the offset gives the location of the current instruction in the routine. This offset is from the starting address of the routine.</td>
</tr>
<tr>
<td></td>
<td>• Compile Date: Contains the year, month and day in which the routine was compiled.</td>
</tr>
<tr>
<td></td>
<td>• Attributes: The available compilation attributes of the compile unit including:</td>
</tr>
<tr>
<td></td>
<td>• A label identifying the LE-supported language such as COBOL, ENT PL/I, C/C++, and so on.</td>
</tr>
<tr>
<td></td>
<td>• Compilation attributes such as EBCDIC, ASCII, IEEE or hexadecimal floating point (HFP). The compilation attributes will only be displayed if there is enough information available.</td>
</tr>
<tr>
<td></td>
<td>• If the CEEDUMP was created under a POSIX environment, POSIX will be displayed.</td>
</tr>
<tr>
<td></td>
<td>The third part of the traceback, which is also referred to as the “Fully Qualified Names” section, contains the following:</td>
</tr>
<tr>
<td></td>
<td>• DSA number</td>
</tr>
<tr>
<td></td>
<td>• Entry</td>
</tr>
<tr>
<td></td>
<td>• Program unit: Similar to the Program Unit column in part 1 except that the server name and the complete program unit (PU) name will be displayed. A PU name will appear here only if it is available to Language Environment.</td>
</tr>
<tr>
<td></td>
<td>• Load Module: The complete pathname of a load module name residing in an UNIX filename will be displayed here if available. The load module's full pathname will be displayed if the PATH environment variable is set such that the pathname of the load module's directory appears before the current directory (.). For load modules found in data sets, the same output shown in the traceback part 1 will also be displayed here.</td>
</tr>
<tr>
<td></td>
<td>The fourth part of the traceback, which is also referred to as the “Full Service Level” section, contains the following:</td>
</tr>
<tr>
<td></td>
<td>• DSA number</td>
</tr>
<tr>
<td></td>
<td>• Entry</td>
</tr>
<tr>
<td></td>
<td>• Service: The full service level string with max length of 64 bytes will be displayed here.</td>
</tr>
</tbody>
</table>
Table 19. Contents of the Language Environment dump  (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [8] Condition Information for Active Routines | Displays the following information for all conditions currently active on the call chain:  
  - Statement showing failing routine and stack frame address of routine  
  - Condition information block (CIB) address  
  - Current condition, in the form of a Language Environment message for the condition raised or a Language Environment abend code, if the condition was caused by an abend  
  - Location: For the failing routine, this is the program unit, entry routine, statement number, and offset.  
  - Machine state, which shows:  
    - Instruction length counter (ILC)  
    - Interruption code  
    - Program status word (PSW)  
    - Contents of 64-bit GPRs 0–15. Note that when the high halves of the registers are not known, they are shown as ********.  
    - Storage dump near condition (2 hex-bytes of storage near the PSW)  
    - Storage pointed to by General Purpose Registers  
    - Contents of access registers, if available  
  This information shows the current values at the time the condition was raised. The high halves of the general registers are dumped, in case they are useful for debugging some applications.  
  If the PSW associated with the condition indicates AMODE 24, the register content will be treated as 24-bit address. |
| [9] Parameters, Registers, and Variables for Active Routines | For each active routine, this section shows:  
  - Routine name and stack frame address  
  - Arguments: For COBOL and Fortran, arguments are shown here rather than with the local variables. For COBOL, arguments are shown as part of local variables. PL/I arguments are not displayed in the Language Environment dump.  
  - Saved registers: This lists the contents of GPRs 0–15 at the time the routine transferred control.  
  - Storage pointed to by the saved registers: Treating the saved contents of each register as an address, 32 bytes before and 64 bytes after the address shown.  
  - Local variables: This section displays the local variables and arguments for the routine. This section also shows the variable type. Variables are displayed only if the symbol tables are available. To generate a symbol table and display variables, use the following compile options:  
    - For COBOL, use TEST(SYM).  
    - For C/C++, use TEST.  
    - For VS COBOL II, use FDUMP.  
    - For COBOL/370, use TEST(SYM).  
    - For COBOL for OS/390 & VM, use TEST(SYM).  
    - For Enterprise COBOL for z/OS V4R2 and prior releases, use TEST(SYM).  
    - For Enterprise COBOL for z/OS V5R1 and later releases, use TEST with any sub options or NOTEST(DWARF).  
    - For Fortran, use SDUMP.  
    - For PL/I, arguments and variables are not displayed. |
<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [10] Control Blocks for Active Routines    | For each active routine controlled by the STACKFRAME option, this section lists contents of related control blocks. The Language Environment-conforming language determines which language-specific control blocks appear. The possible control blocks are:  
  • Stack frame  
  • Condition information block  
  • Language-specific control blocks |
| [11] Storage for Active Routines           | Displays local storage for each active routine. The storage is dumped in hexadecimal, with EBCDIC translations on the right side of the page. There can be other information, depending on the language used. For C/C++ routines, this is the stack frame storage. For COBOL programs, this is language-specific information, WORKING-STORAGE, and LOCAL-STORAGE. |
| [12] Control Blocks Associated with the Thread | Lists the contents of the Language Environment common anchor area (CAA), thread synchronization queue element (SQEL), DLL failure data, and dummy stack frame. Other language-specific control blocks can appear in this section. DLL failure data is described in “Using the DLL failure control block” on page 82. |
| [13] Enclave variables:                    | Displays language specific global variables. This section also shows the variable type. Variables are displayed only if the symbol tables are available.                                                   |
| [14] Enclave Control Blocks                | Lists the contents of the Language Environment enclave data block (EDB) and enclave member list (MEML). The information presented may vary depending on which runtime options are set.  
  • If the POSIX runtime option is set to ON, this section lists the contents of the mutex and condition variable control blocks, the enclave level latch table, and the thread synchronization trace block and trace table.  
  • If DLLs have been loaded, this section shows information for each DLL including the DLL name, load address, use count, writeable static area (WSA) address, and the thread id of the thread that loaded the DLL.  
  • If the HEAPCHK runtime option is set to ON, this section shows the contents of the HEAPCHK options control block (HCOP) and the HEAPCHK element tables (HCEL). A HEAPCHK element table contains the location and length of all allocated storage elements for a heap in the order that they were allocated.  
  • When the call-level suboption of the HEAPCHK runtime option is set, any unfreed storage, which would indicate a storage leak, would be displayed in this area. The traceback could then be used to identify the program which did not free the storage.  
  • If the TRACE runtime option is set to ON, this section shows the contents of the Language Environment trace table. Other language-specific control blocks can appear in this section. |
| [15] Enclave Storage                       | Shows the Language Environment heap storage. For C/C++ and PL/I routines, heap storage is the dynamically allocated storage. For COBOL programs, it is the storage used for WORKING-STORAGE data items. This section also shows the writeable static area (WSA) storage for program objects. Other language-specific storage can appear in this section. |
| [16] File Status and Attributes            | Contains additional information about the file.                                                                                                                                                        |
| [17] Runtime Options Report                | Lists the Language Environment runtime options in effect when the routine was executed.                                                                                                                |
| [18] Process Control Blocks                | Lists the contents for the Language Environment process control block (PCB), process member list (MEML), and if the POSIX runtime option is set to ON, the process level latch table. Other language-specific control blocks can appear in this section. |
Table 19. Contents of the Language Environment dump (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 Additional Language Specific Information</td>
<td>Displays any additional information not included in other sections. For C/C++, it shows the thread ID of the thread that generated the dump and the settings of the errno and errnojr variables for that thread.</td>
</tr>
<tr>
<td>20 CEE3846I CEEDUMP Processing completed.</td>
<td>Identifies the end of the Language Environment dump processing. Similarly, message CEE3845I identifies the start of the dump processing. Message CEE3846I can be used to locate the end of the previous CEEDUMP report when scanning backward in a data set that contains several CEEDUMP reports.</td>
</tr>
</tbody>
</table>

Debugging with specific sections of the Language Environment dump

The following sections describe how you can use particular blocks of the dump to help you debug errors.

**Tracebacks, condition information, and data values section**

The CEE3DMP call with dump options TRACEBACK, CONDITION, and VARIABLES generates output that contains a traceback, information about any conditions, and a list of arguments, registers, and variables. The traceback, condition, and variable information provided in the Language Environment dump can help you determine the location and context of the error without any additional information. The traceback section includes a sequential list for all active routines and the routine name, statement number, and offset where the exception occurred. The condition information section displays a message describing the condition and the address of the condition information block. The arguments, registers, and variables section shows the values of your arrays, structures, arguments, and data during the sequence of calls in your application. Static data values do not appear. Single quotes indicate character fields. These sections of the dump are shown here.

**Upward-growing (non-XPLINK) stack frame section**

The stack frame, also called dynamic save area (DSA), for each active routine is listed in the full dump.

A stack frame chain is associated with each thread in the runtime environment and is acquired every time a separately compiled procedure or block is entered. A stack frame is also allocated for each call to a Language Environment service. All stack frames are back-chained with a stopping stack frame (also called a dummy DSA) as the first stack frame on the stack. Register 13 addresses the recently active stack frame or a standard register save area (RSA). The standard save area back chain must be initialized, and it holds the address of the previous save area. Not all Language Environment-conforming compilers set the forward chain; thus, it cannot be guaranteed in all instances. Calling routines establish the member-defined fields.

When a routine makes a call, registers 0–15 contain the following values:

- R1 is a pointer to parameter list or 0 if no parameter list passed.
- R0, R2–R11 is unreferenced by Language Environment. Caller’s values are passed transparently.
- R12 is the pointer to the CAA if entry to an external routine.
- R13 is the pointer to caller’s stack frame.
- R14 is the return address.
- R15 is the address of the called entry point.
With an optimization level other than 0, C/C++ routines save only the registers used during the running of the current routine. Non-Language Environment RSAs can be in the save area chain. The length of the save area and the saved register contents do not always conform to Language Environment conventions. For a detailed description of stack frames Language Environment storage management, see [z/OS Language Environment Programming Guide](#). Figure 12 shows the format of the upward-growing stack frame.

**Note:** The *Member-defined* fields are reserved for the specific higher level language.

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Flags</td>
</tr>
<tr>
<td>04</td>
<td>CEEDSABACK - Standard Save Area Back Chain</td>
</tr>
<tr>
<td>08</td>
<td>CEEDSAFWD - Standard Save Area Forward Chain</td>
</tr>
<tr>
<td>0C</td>
<td>CEEDSASAVE - GPRs 14, 15, 0-12</td>
</tr>
<tr>
<td>48</td>
<td>Member-defined</td>
</tr>
<tr>
<td>4C</td>
<td>CEEDSANAB - Current Next Available Byte (NAB) in Stack</td>
</tr>
<tr>
<td>50</td>
<td>CEEDSAPNAB - End of Prolog NAB</td>
</tr>
<tr>
<td>54</td>
<td>Member-defined</td>
</tr>
<tr>
<td>58</td>
<td>Member-defined</td>
</tr>
<tr>
<td>5C</td>
<td>Member-defined</td>
</tr>
<tr>
<td>60</td>
<td>Member-defined</td>
</tr>
<tr>
<td>64</td>
<td>Reserved for Debugging</td>
</tr>
<tr>
<td>68</td>
<td>Member-defined</td>
</tr>
<tr>
<td>6C</td>
<td>CEESAMODE - Return Address of the Module That Caused the Last Mode Switch</td>
</tr>
<tr>
<td>70</td>
<td>Member-defined</td>
</tr>
<tr>
<td>74</td>
<td>Member-defined</td>
</tr>
<tr>
<td>78</td>
<td>Reserved for Future Condition Handling</td>
</tr>
<tr>
<td>7C</td>
<td>Reserved for Future Use</td>
</tr>
</tbody>
</table>

**Figure 12. Upward-growing (non-XPLINK) stack frame format**

### Downward-growing (XPLINK) stack frame section

[Figure 13 on page 68](#) shows the format of the downward-growing stack frame. For detailed information about the downward-growing stack, register conventions and parameter passing conventions, see [z/OS Language Environment Programming Guide](#).
Common Anchor Area
Each thread is represented by a common anchor area (CAA), which is the central communication area for Language Environment. All thread- and enclave-related resources are anchored, provided for, or can be obtained through the CAA. The CAA is generated during thread initialization and deleted during thread termination. When calling Language Environment-conforming routines, register 12 points to the address of the CAA.

Use CAA fields as described. Do not modify fields and do not use routine addresses as entry points, except as specified. Fields marked ‘Reserved’ exist for migration of specific languages, or internal use by Language Environment. Language Environment defines their location in the CAA, but not their use. Do not use or reference them except as specified by the language that defines them.

Table 20 on page 69 describes the CAA fields. For more information about the CAA and other structures to which it refers (for example, the DLL failure control block, CEEDLLF), see z/OS Language Environment Vendor Interfaces.

---

**Figure 13. Downward-growing (XPLINK) stack frame format**

<table>
<thead>
<tr>
<th>Low Addresses</th>
<th>Stack Pointer (R4) +2048</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Guard Page (4 KB)</td>
</tr>
<tr>
<td></td>
<td>Stack Frames for called functions</td>
</tr>
<tr>
<td></td>
<td>Backchain Entry Point Return Address</td>
</tr>
<tr>
<td></td>
<td>R8</td>
</tr>
<tr>
<td></td>
<td>R10</td>
</tr>
<tr>
<td></td>
<td>R12</td>
</tr>
<tr>
<td></td>
<td>R14</td>
</tr>
<tr>
<td></td>
<td>+2096</td>
</tr>
<tr>
<td></td>
<td>+2104</td>
</tr>
<tr>
<td></td>
<td>+2108</td>
</tr>
<tr>
<td></td>
<td>+2112</td>
</tr>
<tr>
<td></td>
<td>Local (automatic) Storage Saved FPRs Saved ARs Saved VRs</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>High Addresses</td>
</tr>
</tbody>
</table>

---

Guard Page (4 KB)
Stack Frames for called functions
Backchain Environment Entry Point
Return Address
R8
R9
R10
R11
R12
R13
R14
R15
Reserved (8 bytes)
Debug Area (4 bytes)
Arg Area Prefix (4 Bytes)
Argument Area:
Parm 1
Parm 2
Local (automatic) Storage
Saved FPRs
Saved ARs
Saved VRs

---

<table>
<thead>
<tr>
<th>Stack Pointer (R4) +2048</th>
<th>Guard Page (4 KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Frames for called functions</td>
<td>Backchain Entry Point Return Address</td>
</tr>
<tr>
<td>R8</td>
<td>R9</td>
</tr>
<tr>
<td>+2096</td>
<td>+2104</td>
</tr>
</tbody>
</table>
Table 20. Description of CAA fields

<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
<td>1</td>
<td>CEECAAFLAG0</td>
<td>CAA flag bits, defined as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0–5 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6 CEECAAXHD. A flag used by the condition handler. If the flag is set to 1, the application requires immediate return/percolation to the system on any interrupt or condition handler event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Bit</td>
<td>1</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Bit</td>
<td>1</td>
<td>CEECAALANGP</td>
<td>PL/I language compatibility flags external to Language Environment. The bits are defined as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0–3 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 CEECAATHF. A flag set by PL/I to indicate a PL/I FINISH ON-unit is active. If the flag is set to 1, no PL/I FINISH ON-unit is active. If the flag is set to 0, a PL/I FINISH ON-unit could be active.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5–7 Reserved</td>
</tr>
<tr>
<td>3</td>
<td>Char</td>
<td>5</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Address</td>
<td>4</td>
<td>CEECAABOS</td>
<td>Start of the current storage segment. This field is initially set during thread initialization. It indicates the start of the current stack storage segment. It is altered when the current stack storage segment is changed.</td>
</tr>
<tr>
<td>C</td>
<td>Address</td>
<td>4</td>
<td>CEECAAEOS</td>
<td>This field is used to determine if a stack overflow routine must be called when allocating storage from the user stack. Normally, the value of this field will represent the end of the current user stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the user stack. This field is used by function prologs that do not use FASTLINK linkage conventions.</td>
</tr>
<tr>
<td>10</td>
<td>Char</td>
<td>52</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Signed</td>
<td>2</td>
<td>CEECAATORC</td>
<td>Thread level return code. The thread level return code set by CEESRC callable service.</td>
</tr>
<tr>
<td>46</td>
<td>Signed</td>
<td>2</td>
<td>CEECAATURC</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>Char</td>
<td>44</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>Address</td>
<td>4</td>
<td>CEECAATOVF</td>
<td>Address of stack overflow routine.</td>
</tr>
<tr>
<td>78</td>
<td>Char</td>
<td>168</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>Address</td>
<td>4</td>
<td>CEECAATTN</td>
<td>Address of the Language Environment attention handling routine. The address of the Language Environment attention handling routine supports common runtime environment’s polling code convention for attention processing.</td>
</tr>
<tr>
<td>124</td>
<td>Char</td>
<td>56</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
<td>CAA Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----</td>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>15C</td>
<td>Address</td>
<td>4</td>
<td>CEECAAHLEXIT</td>
<td>Address of the Exit List Control Block set by the HLL user exit CEEBINT.</td>
</tr>
<tr>
<td>160</td>
<td>Char</td>
<td>56</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>198</td>
<td>Bit (96)</td>
<td>12</td>
<td>CEECAAHOOK</td>
<td>Code to pass control to the debugger.</td>
</tr>
<tr>
<td>1A4</td>
<td>Address</td>
<td>4</td>
<td>CEECAADIMA</td>
<td>A(debugger entry)</td>
</tr>
<tr>
<td>1A8</td>
<td>Char</td>
<td>68</td>
<td>CEECAAHOOKS</td>
<td>Hook area. This is the start of 18 fullword execute hooks. Language Environment initializes each fullword to X'07000000'. The hooks can be altered to support various debugging hook mechanisms.</td>
</tr>
<tr>
<td>1A8</td>
<td>Char</td>
<td>4</td>
<td>CEECAAALLOC</td>
<td>ALLOCATE descr. built</td>
</tr>
<tr>
<td>1AC</td>
<td>Char</td>
<td>4</td>
<td>CEECAAASTATE</td>
<td>New statement begins</td>
</tr>
<tr>
<td>1B0</td>
<td>Char</td>
<td>4</td>
<td>CEECAAAENTRY</td>
<td>Block entry</td>
</tr>
<tr>
<td>1B4</td>
<td>Char</td>
<td>4</td>
<td>CEECAAEEXIT</td>
<td>Block exit</td>
</tr>
<tr>
<td>1B8</td>
<td>Char</td>
<td>4</td>
<td>CEECAAMEXIT</td>
<td>Multiple block exit</td>
</tr>
<tr>
<td>1BC</td>
<td>Char</td>
<td>32</td>
<td>CEECAAPATHS</td>
<td>PATH hooks</td>
</tr>
<tr>
<td>1BC</td>
<td>Char</td>
<td>4</td>
<td>CEECAALABEL</td>
<td>At a label constant</td>
</tr>
<tr>
<td>1C0</td>
<td>Char</td>
<td>4</td>
<td>CEECAABCALL</td>
<td>Before CALL</td>
</tr>
<tr>
<td>1C4</td>
<td>Char</td>
<td>4</td>
<td>CEECAACALL</td>
<td>After CALL</td>
</tr>
<tr>
<td>1C8</td>
<td>Char</td>
<td>4</td>
<td>CEECAADO</td>
<td>DO block starting</td>
</tr>
<tr>
<td>1CC</td>
<td>Char</td>
<td>4</td>
<td>CEECAAFTRUE</td>
<td>True part of IF</td>
</tr>
<tr>
<td>1D0</td>
<td>Char</td>
<td>4</td>
<td>CEECAAFALSE</td>
<td>False part of IF</td>
</tr>
<tr>
<td>1D4</td>
<td>Char</td>
<td>4</td>
<td>CEECAAWHEN</td>
<td>WHEN group starting</td>
</tr>
<tr>
<td>1D8</td>
<td>Char</td>
<td>4</td>
<td>CEECAOTHER</td>
<td>OTHERWISE group</td>
</tr>
<tr>
<td>1DC</td>
<td>Char</td>
<td>4</td>
<td>CEECAACGOTO</td>
<td>GOTO hook for C</td>
</tr>
<tr>
<td>1E0</td>
<td>Char</td>
<td>4</td>
<td>CEECAARSVDH1</td>
<td>Reserved hook</td>
</tr>
<tr>
<td>1E4</td>
<td>Char</td>
<td>4</td>
<td>CEECAARSVDH2</td>
<td>Reserved hook</td>
</tr>
<tr>
<td>1E8</td>
<td>Char</td>
<td>4</td>
<td>CEECAAMULTEVT</td>
<td>Multiple Event Hook</td>
</tr>
<tr>
<td>1EC</td>
<td>Bit (32)</td>
<td>4</td>
<td>CEECAAMEVMAK</td>
<td>Multiple Event Hook Mask -End of Debug</td>
</tr>
<tr>
<td>1F0</td>
<td>Char</td>
<td>80</td>
<td>CEECAAMEMBER_AREA</td>
<td></td>
</tr>
<tr>
<td>1F0</td>
<td>Address</td>
<td>4</td>
<td>CEECAACGENE</td>
<td>C/370 CGENE</td>
</tr>
<tr>
<td>1F4</td>
<td>Address</td>
<td>4</td>
<td>CEECAACRENT</td>
<td>C/370 writable static</td>
</tr>
<tr>
<td>1F8</td>
<td>Char</td>
<td>8</td>
<td>CEECAACFLTINIT</td>
<td>Used to convert fixed to float cfltini</td>
</tr>
<tr>
<td>200</td>
<td>Address</td>
<td>4</td>
<td>CEECAACPRMS</td>
<td>Address of parameters passed to main module</td>
</tr>
<tr>
<td>204</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAC_RTL</td>
<td>Combination of 24 unique C/370 trc typ</td>
</tr>
<tr>
<td>208</td>
<td>Address</td>
<td>4</td>
<td>CEECAACTHD</td>
<td></td>
</tr>
<tr>
<td>20C</td>
<td>Address</td>
<td>4</td>
<td>CEECAACURRFECB</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>Address</td>
<td>4</td>
<td>CEECAAEVDV</td>
<td>C/370 vector table</td>
</tr>
<tr>
<td>214</td>
<td>Address</td>
<td>4</td>
<td>CEECAACPCB</td>
<td>Reserved</td>
</tr>
<tr>
<td>218</td>
<td>Address</td>
<td>4</td>
<td>CEECAACEDB</td>
<td>C/370 CEDB</td>
</tr>
<tr>
<td>21C</td>
<td>Char</td>
<td>3</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
Table 20. Description of CAA fields (continued)

<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>21F</td>
<td>Char</td>
<td>1</td>
<td>CEECAASPCFLAG3</td>
<td>Used for SPC</td>
</tr>
<tr>
<td>220</td>
<td>Address</td>
<td>4</td>
<td>CEECAACIO</td>
<td>Address of cio</td>
</tr>
<tr>
<td>224</td>
<td>Char</td>
<td>4</td>
<td>CEECAAFDSETFD</td>
<td>Used by FD_* macros</td>
</tr>
<tr>
<td>228</td>
<td>Char</td>
<td>2</td>
<td>CEECAAFCBMUTEXOK</td>
<td></td>
</tr>
<tr>
<td>22A</td>
<td>Char</td>
<td>2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>22C</td>
<td>Char</td>
<td>4</td>
<td>CEECAATC16</td>
<td></td>
</tr>
<tr>
<td>230</td>
<td>Signed</td>
<td>4</td>
<td>CEECAATC17</td>
<td></td>
</tr>
<tr>
<td>234</td>
<td>Address</td>
<td>4</td>
<td>CEECAAEBCOV</td>
<td>C/370 Open Libvec</td>
</tr>
<tr>
<td>238</td>
<td>Signed</td>
<td>4</td>
<td>CEECAACTOFSV</td>
<td></td>
</tr>
<tr>
<td>23C</td>
<td>Address</td>
<td>4</td>
<td>CEECAATRTPSPACE</td>
<td>C/370 Open Libvec</td>
</tr>
<tr>
<td>240</td>
<td>Char</td>
<td>24</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>258</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASR_USERWORD</td>
<td>TCA Service Rtn Vctr</td>
</tr>
<tr>
<td>25C</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASR_WORKAREA</td>
<td></td>
</tr>
<tr>
<td>260</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASR_GETMAIN</td>
<td></td>
</tr>
<tr>
<td>264</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASR_FREEMAIN</td>
<td></td>
</tr>
<tr>
<td>268</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASR_LOAD</td>
<td></td>
</tr>
<tr>
<td>26C</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASR_DELETE</td>
<td></td>
</tr>
<tr>
<td>270</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASR_EXCEPTION</td>
<td></td>
</tr>
<tr>
<td>274</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASR_ATTENTION</td>
<td></td>
</tr>
<tr>
<td>278</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASR_MESSAGE</td>
<td></td>
</tr>
<tr>
<td>27C</td>
<td>Char</td>
<td>4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>280</td>
<td>Address</td>
<td>4</td>
<td>CEECAALWS</td>
<td>Addr of PL/I LWS</td>
</tr>
<tr>
<td>284</td>
<td>Address</td>
<td>4</td>
<td>CEECAASAVR</td>
<td>Register save</td>
</tr>
<tr>
<td>288</td>
<td>Char</td>
<td>36</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>2AC</td>
<td>Bit</td>
<td>1</td>
<td>CEECAASYSTM</td>
<td>Underlying operating system. The value indicates the operating system supporting the active environment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Undefined. This value should not appear after Language Environment is initialized.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 z/OS</td>
</tr>
<tr>
<td>2AD</td>
<td>Bit (8)</td>
<td>1</td>
<td>CEECAAHRDWR</td>
<td>Underlying hardware. This value indicates the type of hardware on which the routine is running.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Undefined. This value should not appear after Language Environment is initialized.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 System/370, non-XA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 System/370, XA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 System/370, ESA</td>
</tr>
</tbody>
</table>
### Table 20. Description of CAA fields (continued)

<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2AE</td>
<td>Bit (8)</td>
<td>1</td>
<td>CEECAASBSYS</td>
<td>Underlying subsystem. This value indicates the subsystem (if any) on which the routine is running.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0  Undefined. This value should not occur after Language Environment is initialized.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1  Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2  None. The routine is not running under a Language Environment-recognized subsystem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3  TSO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4  IMS™</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5  CICS</td>
</tr>
<tr>
<td>2AF</td>
<td>Bit (8)</td>
<td>1</td>
<td>CEECAFLAGS2</td>
<td>CAA Flag 2, defined as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0  Bimodal addressing is available.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1  Vector hardware is available.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2  Thread terminating.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3  Initial thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4  Library trace is active. The TRACE runtime option was set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5  Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6  CEECAA_ENQ_Wait Interruptible. Thread is in an enqueue wait.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7  Reserved</td>
</tr>
<tr>
<td>2B0</td>
<td>Unsign</td>
<td>1</td>
<td>CEECALEVEL</td>
<td>Language Environment level identifier. This contains a unique value that identifies each release of Language Environment. This number is incremented for each new release of Language Environment.</td>
</tr>
<tr>
<td>2B1</td>
<td>Bit (8)</td>
<td>1</td>
<td>CEECAA_PM</td>
<td>Image of current program mask.</td>
</tr>
<tr>
<td>2B2</td>
<td>Bit (16)</td>
<td>2</td>
<td>CEECA_INVAR</td>
<td>Field that is at the same fixed offset in 31-bit and 64-bit CAAs</td>
</tr>
<tr>
<td>2B3</td>
<td>Bit</td>
<td>1</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>2B4</td>
<td>Address</td>
<td>4</td>
<td>CEECAGETLS</td>
<td>Address of stack overflow for library routines.</td>
</tr>
<tr>
<td>2B8</td>
<td>Address</td>
<td>4</td>
<td>CEECAACELV</td>
<td>Address of the Language Environment library vector. This field is used to locate dynamically loaded Language Environment routines.</td>
</tr>
<tr>
<td>2BC</td>
<td>Address</td>
<td>4</td>
<td>CEECAAGETS</td>
<td>Address of the Language Environment prolog stack overflow routine. The address of the Language Environment get stack storage routine is included in prolog code for fast reference.</td>
</tr>
<tr>
<td>2C0</td>
<td>Address</td>
<td>4</td>
<td>CEECAALBOS</td>
<td>Start of the library stack storage segment. This field is initially set during thread initialization. It indicates the start of the library stack storage segment. It is altered when the library stack storage segment is changed.</td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
<td>CAA Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-----</td>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2C4</td>
<td>Address</td>
<td>4</td>
<td>CEECAALEOS</td>
<td>This field is used to determine if a stack overflow routine must be called when allocating storage from the library stack. Normally, the value of this field will represent the end of the current library stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the library stack. This field is used by function prologs that do not use FASTLINK linkage conventions.</td>
</tr>
<tr>
<td>2C8</td>
<td>Address</td>
<td>4</td>
<td>CEECAALNAB</td>
<td>Next available library stack storage byte. This contains the address of the next available byte of storage on the library stack. It is modified when library stack storage is obtained or released.</td>
</tr>
<tr>
<td>2CC</td>
<td>Address</td>
<td>4</td>
<td>CEECAADMC</td>
<td>Language Environment shunt routine address. Its value is initially set to 0 during thread initialization. If it is nonzero, this is the address of a routine used in specialized exception processing.</td>
</tr>
<tr>
<td>2D0</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAACD</td>
<td>Most recent CAASHAB abend code.</td>
</tr>
<tr>
<td>2D0</td>
<td>Signed</td>
<td>4</td>
<td>CEEAAABCODE</td>
<td>Most recent abend completion code.</td>
</tr>
<tr>
<td>2D4</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAARS</td>
<td>Most recent CAASHAB reason code.</td>
</tr>
<tr>
<td>2D4</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAARSNCODE</td>
<td>Most recent abend reason code.</td>
</tr>
<tr>
<td>2D8</td>
<td>Address</td>
<td>4</td>
<td>CEECAAERR</td>
<td>Address of the current condition information block. After completion of initialization, this always points to a condition information block. During exception processing, the current condition information block contains information about the current exception being processed. Otherwise, it indicates no exception being processed.</td>
</tr>
<tr>
<td>2DC</td>
<td>Address</td>
<td>4</td>
<td>CEECAAGETSX</td>
<td>Address of the user stack extender routine. This routine is called to extend the current stack frame in the user stack. Its address is in the CEECAA for performance reasons.</td>
</tr>
<tr>
<td>2E0</td>
<td>Address</td>
<td>4</td>
<td>CEECAADDSCA</td>
<td>Address of the Language Environment dummy DSA. This address determines whether a stack frame is the dummy DSA, also known as the zeroth DSA.</td>
</tr>
<tr>
<td>2E4</td>
<td>Signed</td>
<td>4</td>
<td>CEECAASECTSZ</td>
<td>Vector section size. This field is used by the vector math services.</td>
</tr>
<tr>
<td>2E8</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAPARTSUM</td>
<td>Vector partial sum number. This field is used by the vector math services.</td>
</tr>
<tr>
<td>2EC</td>
<td>Signed</td>
<td>4</td>
<td>CEECAASSEXPNTE</td>
<td>Log of the vector section size. This field is used by the vector math services.</td>
</tr>
<tr>
<td>2F0</td>
<td>Address</td>
<td>4</td>
<td>CEECAAEDB</td>
<td>Address of the Language Environment EDB. This field points to the encompassing EDB.</td>
</tr>
<tr>
<td>2F4</td>
<td>Address</td>
<td>4</td>
<td>CEECAAPCB</td>
<td>Address of the Language Environment PCB. This field points to the encompassing PCB.</td>
</tr>
<tr>
<td>2F8</td>
<td>Address</td>
<td>4</td>
<td>CEECAAEYEPTRE</td>
<td>Address of the CAA eye catcher. The CAA eye catcher is CEECAA. This field can be used for validation of the CAA.</td>
</tr>
</tbody>
</table>

Chapter 3. Using Language Environment debugging facilities 73
<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2FC</td>
<td>Address</td>
<td>4</td>
<td>CEECAAPTR</td>
<td>Address of the CAA. This field points to the CAA itself and can be used in validation of the CAA.</td>
</tr>
<tr>
<td>300</td>
<td>Address</td>
<td>4</td>
<td>CEECAAGETS1</td>
<td>Non-DSA stack overflow. This field is the address of a stack overflow routine, which cannot guarantee that the current register 13 is pointing at a stack frame. Register 13 must point, at a minimum, to a save area.</td>
</tr>
<tr>
<td>304</td>
<td>Address</td>
<td>4</td>
<td>CEECAASHAB</td>
<td>ABEND shunt routine. Its value is initially set to zero during thread initialization. If it is nonzero, this is the address of a routine used in specialized exception processing for ABENDs that are intercepted in the ESTAE exit.</td>
</tr>
<tr>
<td>308</td>
<td>Address</td>
<td>4</td>
<td>CEECAAPRGCK</td>
<td>Routine interrupt code for CEECAADMC. If CEECAADMC is nonzero, and a routine interrupt occurs, this field is set to the routine interrupt code and control is passed to the address in CEECAAMDC.</td>
</tr>
<tr>
<td>30C</td>
<td>Bit (8)</td>
<td>1</td>
<td>CEECAFLAG1</td>
<td>CAA flag bits, defined as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 - 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>30D</td>
<td>Char</td>
<td>1</td>
<td>CEECAASHAB_KEY</td>
<td>IPK result when CEECAASHAB is set.</td>
</tr>
<tr>
<td>30E</td>
<td>Char</td>
<td>2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>310</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAURC</td>
<td>Thread level return code. This is the common place for members to set the return codes for subroutine-to-subroutine return code processing.</td>
</tr>
<tr>
<td>314</td>
<td>Address</td>
<td>4</td>
<td>CEECAAES</td>
<td>Determine if a stack overflow routine must be called when allocating storage from the user stack. Normally, the value of this field will represent the end of the current user stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the user stack. This field is used by function prologs that use FASTLINK linkage conventions.</td>
</tr>
</tbody>
</table>
### Table 20. Description of CAA fields (continued)

<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>Address</td>
<td>4</td>
<td>CEECAALESS</td>
<td>Determine if a stack overflow routine must be called when allocating storage from the library stack. Normally, the value of this field will represent the end of the current library stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the library stack. This field is used by function prologs that use FASTLINK linkage conventions.</td>
</tr>
<tr>
<td>31C</td>
<td>Address</td>
<td>4</td>
<td>CEECAAOGETS</td>
<td>Overflow from user stack allocations.</td>
</tr>
<tr>
<td>320</td>
<td>Address</td>
<td>4</td>
<td>CEECAAOGETLS</td>
<td>Overflow from library stack allocations.</td>
</tr>
<tr>
<td>324</td>
<td>Address</td>
<td>4</td>
<td>CEECAAPICICB</td>
<td>Address of the preinitialization compatibility control block.</td>
</tr>
<tr>
<td>328</td>
<td>Address</td>
<td>4</td>
<td>CEECAAOGETSX</td>
<td>User DSA exit from OPLINK.</td>
</tr>
<tr>
<td>32C</td>
<td>Signed</td>
<td>2</td>
<td>CEECAAGOSMR</td>
<td>Go some more—Used CEEHTRAV multiple.</td>
</tr>
<tr>
<td>32E</td>
<td>Signed</td>
<td>2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>330</td>
<td>Address</td>
<td>4</td>
<td>CEECAALEOV</td>
<td>This field is the address of the Language Environment library vector for z/OS UNIX support.</td>
</tr>
<tr>
<td>334</td>
<td>Signed</td>
<td>4</td>
<td>CEECAA_SIGSCTR</td>
<td>SIGSAFE counter.</td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
<td>CAA Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-----</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>338</td>
<td>Bit (32)</td>
<td>4</td>
<td>CEECAA_SIGSFLG</td>
<td>SIGSAFE flags indicate the signal safety of the library and are defined, as follows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0  CEECAA_SIGPUTBACK. The signal cannot be delivered, therefore the signal is put back to the kernel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1  CEECAA_SA_RESTART. Indicates that a signal registered with the SA_RESTART flag interrupted the last kernel call, and the signal catcher returned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2  Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3  CEECAA_SIGSAFE. It is safe to deliver the signal, while in library code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4  CEECAA_CANCELSAFE. It is safe to deliver the cancel signal, while in library code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5  CEECAA_SIGRESYNCH. CEECAA_sigsynch flag was on last time CEEOSIGR resolicited a signal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6  CEECAA_FRZ_UNSAFE. This thread is in an unsafe state to be frozen.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7  CEECAA_NOAPPREGS. User application registers may be saved in a nonstandard place.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8  CEECAA_EINTR_RSOL. Secondary Signal resolicitation is in progress, after EINTR errno from inner function.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9  CEECAA_EINTR_PTB. Secondary resolicited signal has been put back.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 CEECAA_EINTR_REST. User signal catcher returned after catching secondary resolicited signal with SA_RESTART in effect.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 CEECAA_EINTR_SIGG. Stray signal interrupted CEEOSIGG while secondary signal resolicitation was in progress.</td>
</tr>
<tr>
<td>33A</td>
<td>Bit (16)</td>
<td>2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>33C</td>
<td>Char</td>
<td>8</td>
<td>CEECAATHDID</td>
<td>This field is the thread identifier</td>
</tr>
<tr>
<td>344</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_DCRENT</td>
<td>Reserved</td>
</tr>
<tr>
<td>348</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_DANCHOR</td>
<td>Reserved</td>
</tr>
<tr>
<td>34C</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TOCTOC</td>
<td>TOC anchor for CREN.</td>
</tr>
<tr>
<td>354</td>
<td>Signed</td>
<td>4</td>
<td>CEECAACICSRSN</td>
<td>CICS reason code from member language.</td>
</tr>
<tr>
<td>358</td>
<td>Address</td>
<td>4</td>
<td>CEECAAAMEMBR</td>
<td>Address of thread-level member list.</td>
</tr>
<tr>
<td>35C</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_SIGNAL_STATUS</td>
<td>Signal status of the terminating thread member list.</td>
</tr>
<tr>
<td>360</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_HCOM_REG7</td>
<td>HCOM saved R7.</td>
</tr>
<tr>
<td>360</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_HCOM_REG14</td>
<td>HCOM saved R14.</td>
</tr>
<tr>
<td>364</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_STACKFLOOR</td>
<td>Lowest usable address in XP stack.</td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
<td>CAA Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>368</td>
<td>Address</td>
<td>4</td>
<td>CEECAAHPGETS</td>
<td>XP stack extension rtn.</td>
</tr>
<tr>
<td>36C</td>
<td>Address</td>
<td>4</td>
<td>CEECAAECHPXV</td>
<td>C/C++ XPLINK libvec.</td>
</tr>
<tr>
<td>370</td>
<td>Address</td>
<td>4</td>
<td>CEECAAFOR1</td>
<td>Reserved for FORTRAN.</td>
</tr>
<tr>
<td>374</td>
<td>Address</td>
<td>4</td>
<td>CEECAAFOR2</td>
<td>Reserved for FORTRAN.</td>
</tr>
<tr>
<td>378</td>
<td>Address</td>
<td>4</td>
<td>CEECAATHREADHEAPID</td>
<td>Thread heap ID.</td>
</tr>
<tr>
<td>37C</td>
<td>Signed</td>
<td>4</td>
<td>CEECA_SYSTNCODE</td>
<td>System (kernel) return code.</td>
</tr>
<tr>
<td>380</td>
<td>Signed</td>
<td>4</td>
<td>CEECA_SYS_RSNCODE</td>
<td>System (kernel) reason code.</td>
</tr>
<tr>
<td>384</td>
<td>Address</td>
<td>4</td>
<td>CEECAAGETFN</td>
<td>Address of the WSA swap routine.</td>
</tr>
<tr>
<td>388</td>
<td>Address</td>
<td>4</td>
<td>CEECA_JIT1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>38C</td>
<td>Address</td>
<td>4</td>
<td>CEECA_JIT2</td>
<td>Reserved.</td>
</tr>
<tr>
<td>390</td>
<td>Address</td>
<td>4</td>
<td>CEECAASIGNGPTR</td>
<td>Pointer to 'signam' external variable in a C application.</td>
</tr>
<tr>
<td>394</td>
<td>Signed</td>
<td>4</td>
<td>CEECAASIGNG</td>
<td>Value of sign of lgamma() -1 - negative sign 0 - zero +1 - positive sign.</td>
</tr>
<tr>
<td>398</td>
<td>Address</td>
<td>4</td>
<td>CEECA_FORDBG</td>
<td>Ptr to AFHDBHIM - FORTRAN hook interface.</td>
</tr>
<tr>
<td>39C</td>
<td>Bit (8)</td>
<td>1</td>
<td>CEECAAB_STATUS</td>
<td>Validity flags.</td>
</tr>
<tr>
<td>39D</td>
<td>Unsign</td>
<td>1</td>
<td>CEECA_STACKDIRECTION</td>
<td>Stack direction.</td>
</tr>
<tr>
<td>39E</td>
<td>Bit</td>
<td>2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3A0</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAB_GR0</td>
<td>Reg 0 at the time of abend.</td>
</tr>
<tr>
<td>3A4</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAB_ICD1</td>
<td>SDWAICD1.</td>
</tr>
<tr>
<td>3A8</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAB_ABCC</td>
<td>SDWAABCC.</td>
</tr>
<tr>
<td>3AC</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAB_CRC</td>
<td>SDWACRC.</td>
</tr>
<tr>
<td>3B0</td>
<td>Address</td>
<td>4</td>
<td>CEECAAGTS</td>
<td>Entry point of CEEVAGTS routine.</td>
</tr>
<tr>
<td>3B4</td>
<td>Address</td>
<td>4</td>
<td>CEECA_LER5N1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>3B8</td>
<td>Address</td>
<td>4</td>
<td>CEECAHEREP</td>
<td>Address of CEEHERP routine.</td>
</tr>
<tr>
<td>3BC</td>
<td>Address</td>
<td>4</td>
<td>CEECAUSTKBOB</td>
<td>Start of user stack segment.</td>
</tr>
<tr>
<td>3C0</td>
<td>Address</td>
<td>4</td>
<td>CEECAUSTKEOS</td>
<td>End of user stack segment.</td>
</tr>
<tr>
<td>3C4</td>
<td>Address</td>
<td>4</td>
<td>CEECAUSERRTN@</td>
<td>Address of thread start routine. Undefined on IPT or prior to thread init event.</td>
</tr>
<tr>
<td>3C8</td>
<td>Bit</td>
<td>8</td>
<td>CEECAAUDHOOK</td>
<td>Hook swapping XPLINK.</td>
</tr>
<tr>
<td>3D0</td>
<td>Address</td>
<td>4</td>
<td>CEECAACEL_HPXV_B</td>
<td>Address of XPLINK compat vector for Base library.</td>
</tr>
<tr>
<td>3D4</td>
<td>Address</td>
<td>4</td>
<td>CEECAACEL_HPXV_M</td>
<td>Address of XPLINK compat vector for Math library.</td>
</tr>
<tr>
<td>3D8</td>
<td>Address</td>
<td>4</td>
<td>CEECAACEL_HPXV_L</td>
<td>Address of XPLINK compat vector for Locale library.</td>
</tr>
<tr>
<td>3DC</td>
<td>Address</td>
<td>4</td>
<td>CEECAACEL_HPXV_O</td>
<td>Address of XPLINK compat vector for Open library.</td>
</tr>
<tr>
<td>3E0</td>
<td>Address</td>
<td>4</td>
<td>CEECAACEL4VEC3</td>
<td>Address of 3rd C-RTL library vector.</td>
</tr>
<tr>
<td>3E4</td>
<td>Address</td>
<td>4</td>
<td>CEECA_CEEDLLF</td>
<td>Address of the newest CEEDLLF control block.</td>
</tr>
<tr>
<td>3E8</td>
<td>Address</td>
<td>4</td>
<td>CEECA_SAVSTACK</td>
<td>Zero or saved stack pointer. This field can be used to save the stack pointer before calling a routine with OS_NOSTACK linkage. After the call returns, this field must be set back to zero.</td>
</tr>
<tr>
<td>3EC</td>
<td>Char</td>
<td>4</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
Table 20. Description of CAA fields (continued)

<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3F0</td>
<td>Char</td>
<td>4</td>
<td>CEECAA_USER_WORD</td>
<td>4-byte user field available for application use. In pre-initialization (CEEPIPI) environments, this field is initialized in the IPT CAA from the CEEPIPI set_user_word function. This field is initialized to 0 in non-CEEPIPI environments (including all nested enclaves), and for all non-IPT CAAs in CEEPIPI environments. This field is not otherwise accessed by Language Environment.</td>
</tr>
<tr>
<td>3F4</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_SAVSTACK_ASYNC</td>
<td>Zero or address of field that is zero or saved stack pointer. An application that has large sections of code that do not require access to the Language Environment stack but could benefit from having an additional register available can use this field.</td>
</tr>
<tr>
<td>3F8</td>
<td>Char</td>
<td>4</td>
<td>CEECAA_STACK_GUARD</td>
<td>Zero or Stack Guard token.</td>
</tr>
</tbody>
</table>

**Condition information block**

[Figure 14 on page 79](#) shows the condition information block. The Language Environment condition manager creates a condition information block (CIB) for each condition that is encountered in the Language Environment environment. The CIB holds data required by the condition handling facilities and pointers to locations of other data. The address of the current CIB is in the CAA.

For COBOL, Fortran, and PL/I applications, Language Environment provides macros (in the SCEESAMP data set) that map the CIB. For C/C++ applications, the macros are in `leaw.h`. 
### Figure 14. Condition information block (Part A)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>Condition Information Block</td>
</tr>
<tr>
<td></td>
<td>Eye catcher</td>
</tr>
<tr>
<td>+4</td>
<td>Previous Condition Information Block</td>
</tr>
<tr>
<td>+8</td>
<td>Most Recent Condition Information Block</td>
</tr>
<tr>
<td>+C</td>
<td>Size of Condition Information Block</td>
</tr>
<tr>
<td></td>
<td>Version of Condition Information Block</td>
</tr>
<tr>
<td>+10</td>
<td>Platform Identifier</td>
</tr>
<tr>
<td></td>
<td>3 = Language Environment</td>
</tr>
<tr>
<td>+18</td>
<td>Current Language Environment Condition</td>
</tr>
<tr>
<td>+24</td>
<td>Address of Machine State</td>
</tr>
<tr>
<td></td>
<td>Time of Interrupt</td>
</tr>
<tr>
<td>+28</td>
<td>Previous Language Environment Condition</td>
</tr>
<tr>
<td>+37</td>
<td>Condition Flags</td>
</tr>
<tr>
<td>+38</td>
<td>Handle Cursor</td>
</tr>
<tr>
<td>+44</td>
<td>Resume Cursor</td>
</tr>
<tr>
<td>+54</td>
<td>Physical Callee Stack Frame Pointer (handle cursor)</td>
</tr>
<tr>
<td>+58</td>
<td>DSA format for Stack frame in the Handle Cursor</td>
</tr>
<tr>
<td></td>
<td>0 = non-XPLINK</td>
</tr>
<tr>
<td></td>
<td>1 = XPLINK</td>
</tr>
<tr>
<td>+59</td>
<td>DSA format for Physical CalleeStack frame (handle cursor) (reserved)</td>
</tr>
<tr>
<td>+5A</td>
<td>(reserved)</td>
</tr>
<tr>
<td>+5B</td>
<td>Address of recorded dump dataset name (reserved)</td>
</tr>
<tr>
<td>+68</td>
<td>Address of recorded dump dataset name</td>
</tr>
<tr>
<td>+6C</td>
<td>(reserved)</td>
</tr>
<tr>
<td>+B0</td>
<td>Status Flag 5</td>
</tr>
<tr>
<td>+B1</td>
<td>Status Flag 6</td>
</tr>
<tr>
<td>+B2</td>
<td>Status Flag 7</td>
</tr>
<tr>
<td>+B4</td>
<td>Abend Code Word</td>
</tr>
</tbody>
</table>

- **Status Flag 5**
  - 64 = An SDWA is Associated with the Condition
  - 0  = Storage Condition

- **Abend Code Word**

---

Chapter 3. Using Language Environment debugging facilities 79
The flags for Condition Flag 4:

- 2 The resume cursor was moved.
- 4 The message service processed the condition.
- 8 The resume cursor was explicitly moved.

The flags for Status Flag 5, Language Environment events:

- 1 Caused by an attention interrupt.
- 2 Caused by a signaled condition.
- 4 Caused by a promoted condition.
- 8 Caused by a condition management raised TIU.
- 32 Caused by a condition signaled via CEEOKILL. The signaled-via-CEEOKILL flag is always set with the signaled flag; thus, a signaled condition.
condition can have a value of either 2 or 34. (The value is 2 if the signaled condition does not come through CEEOKILL. If it comes through CEEOKILL, its value is 2+32=34.)

64 Caused by a program check.
128 Caused by an abend.

The flags for Status Flag 6, Language Environment actions:
2 Doing stack frame zero scan.
4 H-cursor pointing to owning SF.
8 Enable only pass (no condition pass).
16 MRC type 1.
32 Resume allowed.
64 Math service condition.
128 Abend reason code valid.

**Address of recorded dump data set name:** If this address is not 0, then it points to a 44-byte fixed-length character string. If the length of the data set name is less than 44, the character string is EBCDIC-encoded and is padded by blanks.

The language-specific function codes for the CIB:
X'1' For condition procedure.
X'2' For enablement.
X'3' For stack frame zero conditions.

**Using the machine state information block**
The Language Environment machine state information block contains condition information pertaining to the hardware state at the time of the error. Figure 16 on page 82 shows the machine state information block.
Using the DLL failure control block

The CEEDLLF control block contains error diagnostics corresponding to an implicit or explicit DLL failure. Diagnostics describing up to 10 of the most recent DLL failures are available in a circular list of CEEDLLF control blocks. When viewing a dump, the in-use CEEDLLF control blocks are displayed from newest to oldest. See “Understanding the Language Environment IPCS VERBEXIT LEDATA output” on page 93 for the contents of CEEDLLF fields.
Multiple enclave dumps

Figure 17 on page 84 illustrates the information available in the Language Environment dump and the order of information for multiple enclaves. If multiple enclaves are used, the dump service generates data and storage information for the most current enclave and moves up the chain of enclaves to the starting enclave in a LIFO order. For example, if two enclaves are used, the dump service first generates output for the most current enclave. Then the service creates output for the previous enclave. A thread terminating in a non-POSIX environment is analogous to an enclave terminating because Language Environment Version 1 supports only single threads.
If multiple nested CEEPIPI Main-DP environments are present, the dump service generates data and storage information for the most current Main-DP environment and moves up the chain of Main-DP environments to the starting Main-DP environment in LIFO order.

Figure 17. Language Environment dump of multiple enclaves
When multiple nested CEEPIPI Main-DP environments are present in the dump output, the information in Figure 17 on page 84 appears for the most current Main-DP environment. For the other chained Main-DP environments, only the traceback section appears. The following is an example:

**** Information for CEEPIPI token xxxxxxxx ****

- information for newest enclave
- information for next older enclave
- information for oldest enclave

Other information

**** Information for CEEPIPI token xxxxxxxx ****

- traceback for newest enclave
- traceback for next older enclave
- traceback for next older enclave

**** Information for CEEPIPI token xxxxxxxx ****

- traceback for newest enclave
- traceback for next older enclave
- traceback for next older enclave

**** Information for CEEPIPI token xxxxxxxx ****

- traceback for newest enclave
- traceback for next older enclave

Generating a system dump

A system dump contains the storage information needed to diagnose errors. You can use Language Environment to generate a system dump through any of the following methods:

**DYNDUMP(hlq,DYNAMIC,TDUMP)**

You can use the DYNDUMP runtime option to obtain IPCS-readable dumps of user applications that would ordinarily be lost due to the absence of a SYSMDUMP, SYSUDUMP, or SYSABEND DD statement.

**TERMTHDACT(UAONLY, UATRACE, or UADUMP)**

You can use these runtime options, with TRAP(ON), to generate a system dump if an unhandled condition of severity 2 or greater occurs. For more details about the level of dump information produced by each of the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 42.

**TRAP(ON,NOSPIE) TERMTHDACT(UAIMM)**

TRAP(ON,NOSPIE) TERMTHDACT(UAIMM) generates a system dump of the user address space of the original abend or program interrupt prior to the Language Environment condition manager processing the condition.

**ABPERC(abcode)**

The ABPERC runtime option specifies one abend code that is exempt from the Language Environment condition handler. The Language Environment condition handler percolates the specified abend code to the operating system. The operating system handles the abend and generates a system dump. ABPERC is ignored under CICS.

**Abend Codes in Initialization Assembler User Exit**

Abend codes listed in the initialization assembler user exit are passed to the operating system. The operating system can then generate a system dump.
CEE3ABD

You can use the CEE3ABD callable service to cause the operating system to handle an abend.

See system or subsystem documentation for detailed system dump information.

The method for generating a system dump varies for each of the Language Environment runtime environments. The following sections describe the recommended steps needed to generate a system dump in a batch, IMS, CICS, and z/OS UNIX shell runtime environments. Other methods may exist, but these are the recommended steps for generating a system dump.

For details on setting Language Environment runtime options, see {z/OS Language Environment Programming Guide}

Steps for generating a system dump in a batch runtime environment

Perform the following steps to generate a system dump in a batch runtime environment. When you are done, you will have generated a system dump in a batch runtime environment.

1. Specify runtime options TERMTHDACT(UAONLY, UADUMP, UATRACE, or UAIMM), and TRAP(ON). If you specify the suboption UAIMM then you must set TRAP(ON,NOSPIE). The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details on the TERMTHDACT suboptions, see "Generating a Language Environment dump with TERMTHDACT" on page 42.

2. Decide whether to include a SYSMDUMP DD card or use the DYNDUMP runtime option.
   - Include a SYSMDUMP DD card with the desired data set name and DCB information:

   ```
   LRECL=4160, BLKSIZE=4160, and RECFM=FBS.
   ```

   - Specify the DYNDUMP runtime option with the following information:

   ```
   DYNDUMP (hlq,DYNAMIC,TDUMP)
   ```

3. Rerun the program.

Steps for generating a system dump in an IMS runtime environment

Perform the following steps to generate a system dump in an IMS runtime environment. When you are done, you will have generated system dump in an IMS runtime environment.

1. Specify runtime options TERMTHDACT(UAONLY, UADUMP, UATRACE, or UAIMM), ABTERM(ABEND), and TRAP(ON). If you specify the suboption UAIMM, then you must set TRAP(ON,NOSPIE). The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details on the TERMTHDACT suboptions, see "Generating a Language Environment dump with TERMTHDACT" on page 42.
2. Decide whether to include a SYSMDUMP DD card or use the DYNDUMP runtime option.
   - Include a SYSMDUMP DD card with the desired data set name and DCB information:
     
     LRECL=4160, BLKSIZE=4160, and RECFM=FBS.
     
     - Specify the DYNDUMP runtime option with the following information:

     DYNDUMP (hiq,DYNAMIC,TDUMP)

3. Rerun the program.

**Steps for generating a system dump in a CICS runtime environment**

**Before you begin:** Under CICS, a system dump provides the most useful information for diagnosing problems. However, if you have a Language Environment U4038 abend, CICS will not generate a system dump. To generate diagnostic information for a CICS runtime environment with a Language Environment U4038 abend, you must create a Language Environment U4039 abend. For instructions on how to create a Language Environment U4039 abend, see “Steps for generating a Language Environment U4039 abend.”

**Note:** DYNDUMP is ignored in a CICS environment.

Perform the following steps to generate a system dump in a CICS runtime environment. When you are done, you will have generated a system dump in a CICS runtime environment.

1. Specify runtime options TERMTHDACT(UAONLY, UADUMP, or UATRACE), ABTERM(ABEND), and TRAP(ON). The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For more details on the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 42.

2. Update the transaction dump table with the CICS-supplied CEMT command:

   CEMT SET TRD(40XX) SYS ADD

   **Result** You will see CEMT output.

   **Example**

   STATUS: RESULTS - OVERTYPE TO MODIFY
   Trd(4088) Sys Loc Max( 999 ) Cur(0000)

3. Rerun the program.

**Steps for generating a Language Environment U4039 abend**

If you have a Language Environment U4038 abend, CICS will not generate a system dump. To generate diagnostic information, you must create a Language
Environment U4039 abend by performing the following steps. By setting these runtime options, a Language Environment U4039 abend occurs which generates a system dump.

1. Specify DUMP=YES in CICS DFHSIT.
2. Specify runtime options TERMTHDACT(UAONLY, UATRACE, or UADUMP), ABTERM(ABEND), and TRAP(ON)
3. Rerun the program.

Steps for generating a system dump in a z/OS UNIX shell

Perform the following steps to generate a system dump from a z/OS UNIX shell:
• Using _BPXK_MDUMP
1. Specify where to write the system dump
   - To write the system dump to a z/OS data set, issue the following command, where filename is a fully qualified data set name with DCB information: LRECL=4160, BLKSIZE=4160, and RECFM=FBS.

   ```
   export _BPXK_MDUMP=filename
   ```

   Example
   ```
   export _BPXK_MDUMP=hlq.mydump
   ```
   - To write the system dump to an HFS file, issue the following command, where filename is a fully qualified HFS filename.

   ```
   export _BPXK_MDUMP=filename
   ```

   Example
   ```
   export _BPXK_MDUMP=/tmp/mydump.dmp
   ```
   2. Specify Language Environment runtime options, where suboption is UAONLY, UADUMP, UATRACE, or UAIMM.

   ```
   export _CEE_RUNOPTS="termthdact(suboption)"
   ```

   If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For more details about the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 42.

3. Rerun the program.

When you are done, the system dump is written to the data set name or HFS file name specified. For additional _BPXK_MDUMP information see z/OS UNIX System Services Planning and z/OS UNIX System Services Programming: Assembler Callable Services Reference.

• Using DYNDUMP
1. Specify Language Environment runtime options:

   ```
   export _CEE_RUNOPTS="termthdact(suboption),DYNDUMP(hlq,DYNAMIC,TDUMP)"
   ```

   suboption is UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set,
TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For more details about the TERMTHDACT suboptions, see "Generating a Language Environment dump with TERMTHDACT" on page 42.

hlq the high level qualifier for the dump data set to be created.

2. Rerun the program.

When you are done, the system dump is written to the name generated by the DYNDUMP runtime option. For additional DYNDUMP information see z/OS Language Environment Programming Reference.

Note: You can also specify the signal SIGDUMP on the kill command to generate a system dump of the user address space. For more information regarding the SIGDUMP signal, see z/OS UNIX System Services Command Reference.

Formatting and analyzing system dumps

You can use the interactive problem control system (IPCS) to format and analyze system dumps. Language Environment provides an IPCS VERBEXIT LEDATA that can be used to format Language Environment control blocks. For more information on using IPCS, see z/OS MVS IPCS User’s Guide.

Preparing to use the Language Environment support for IPCS

Use the following guidelines before you use IPCS to format Language Environment control blocks:

- Ensure that your IPCS job can find the CEEIPCSP member.

  IPCS provides an exit control table with imbed statements to enable other products to supply exit control information. The IPCS default table, BLSCECT, normally in the SYS1.PARMLIB library, has the following entry for Language Environment:

  IMBED MEMBER(CEEIPCSP) ENVIRONMENT(IPCS)

  The Language Environment-supplied CEEIPCSP member, installed in the SYS1.PARMLIB library, contains the Language Environment-specific entries for the IPCS exit control table.

- Provide an IPCSPARM DD statement to specify the libraries containing the IPCS control tables.

  Example

  //IPCSPARM DD DSN=SYS1.PARMLIB,DISP=SHR

  - Ensure that your IPCS job can find the Language Environment-supplied ANALYZE exit routines installed in the SYS1.MIGLIB library.

  - To aid in debugging system or address space hang situations, Language Environment mutexes, latches and condition variables can be displayed if the CEEIPCSP member you are using is updated to identify the Language Environment ANALYZE exit, by including the following statement:

  EXIT EP(CEEEEANLZ) ANALYZE
Understanding Language Environment IPCS VERBEXIT – LEDATA

Purpose

Use the LEDATA verb exit to format data for Language Environment. This VERBEXIT provides information about the following topics:

- A summary of Language Environment at the time of the dump
- Runtime Options
- Storage Management Control Blocks
- Condition Management Control Blocks
- Message Handler Control Blocks
- C/C++ Control Blocks
- COBOL Control Blocks
- PL/I Control Blocks

Format

```
VERBEXIT LEDATA ['parameter[,parameter]...']
```

Report Type Parameters:

- `SUM`  
- `HEAP | STACK | SM`  
- `HPT(number) [ HPTTTCB(address) ] [ HPTCELL(address) ] [ HPTLOC(location) ] ]`  
- `CM ]`  
- `HH ]`  
- `CEEDUMP ]`  
- `COMP(value) ]`  
- `PTBL(value) ]`  
- `ALL ]`

Data Selection Parameters:

- `DETAIL | EXCEPTION ]`

Control Block Selection Parameters:

- `CAA(caa-address) ]`  
- `DSA(dsa-address) ]`  
- `TCB(tcb-address) ]`  
- `ASID(address-space-id) ]`  
- `NTHREADS(value) ]`

Parameters

The following sections describe the different types of supported parameters. Note that only hexadecimal characters can be specified as addresses provided in LEDATA parameters. Special characters cause the formatter to fail. Therefore, to specify a 64-bit address as a parameter, it must be in the form like 123456789 instead of 1_23456789.

Report type parameters

Use the following parameters to select the type of report. You can specify as many reports as you wish. If you omit these parameters, the default is SUMMARY.

**SUMmary**

Requests a summary of the Language Environment at the time of the dump. The following information is included:

- TCB address
- Address Space Identifier
- Language Environment Release
- Active members
- Formatted CAA, PCB, RCB, EDB and PMCB
- Runtime Options in effect

HEAP | STACK | SM

**HEAP**
Requests a report on Storage Management control blocks pertaining to HEAP storage, as well as a detailed report on heap segments. The detailed report includes information about the free storage tree in the heap segment, and information about each allocated storage element. It also specifies a heap pools report with information useful to find potential damaged cells. Note that Language Environment does not provide support for alternative Vendor Heap Manager (VHM) data.

**STACK**
Requests a report on Storage Management control blocks pertaining to STACK storage.

**SM** Requests a report on Storage Management control blocks. This is the same as specifying both HEAP and STACK.

**HPT(number)** [ HPTTCB(address) ] [ HPTCELL(address) ] [ HPTLOC(location) ]

**HPT(number)**
Requests that the HEAPPOOLS trace, if available, be formatted. If the value is 0 or *, the trace for every HEAPPOOLS pool id is formatted. If the value is a single number (1-12), the trace for the specific HEAPPOOLS pool id is formatted. If only the HPT keyword is specified with no value, the trace behaves similar to when the value is *. If no filter is specified, all of the entries are formatted for the specific pool id.

**HPTTCB(address)**
Filters the HEAPPOOLS trace table, if available, printing only those entries for a given TCB address (address).

**HPTCELL(address)**
Filters the HEAPPOOLS trace table, if available, printing only those entries for a given cell address (address).

**HPTLOC(location)**
Filters the HEAPPOOLS trace table, if available, printing only those entries for a given virtual storage location (location). The following values are valid:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Display entries located in virtual storage below the bar.</td>
</tr>
<tr>
<td>64</td>
<td>Display entries located in virtual storage above the bar.</td>
</tr>
<tr>
<td>ALL</td>
<td>Display entries located in virtual storage below or above the bar.</td>
</tr>
</tbody>
</table>

**Note:**
1. Filter options without specifying HPT implies HPT(*).
2. You can specify multiple options together, like HPTTCB and HPTCELL. All pieces of information must match the trace entry for it to be formatted. If location and cell contradict each other, such as HPTLOC(31) and HPTCELL(64bit addr), an error will be displayed.

**CM** Requests a report on Condition Management control blocks.
MH  Requests a report on Message Handler control blocks.

CEEDump
Requests a CEEDUMP-like report. Currently this includes the traceback, the
Language Environment trace, and thread synchronization control blocks at
process, enclave and thread levels.

If the dump output has multiple nested enclaves or multiple nested CEEPIPI
Main-DP environments, tracebacks will appear for each enclave in each
Main-DP environment. This is similar to how the tracebacks appear in the
CEEDUMP output. See the section “Multiple enclave dumps” on page 83 for a
description of CEEDUMP output when multiple enclave and Main-DP
environments are present.

PTBL(value)
Requests that PreInit tables be formatted according to the following values:

CURRENT
   If current is specified, the PreInit table associated with the current or
   specified TCB is displayed.

address
   If an address is specified, the PreInit table at that address is specified.

* All active and dormant PreInit tables within the current address space are
displayed; this option is time-consuming.

ACTIVE
   The PreInit tables for all TCBs in the address space are displayed.

COMP(value)
Requests component control blocks to be formatted according to the following
values:

C  Requests a report on C/C++ runtime control blocks.

CIO  Requests a report on C/C++ I/O control blocks.

COBOL  Requests a report on COBOL-specific control blocks.

PLI  Requests a report on PL/I-specific control blocks.

ALL  Requests a report on all the preceding control blocks.

If the value specified in COMP is not one of the values (C, CIO, COBOL, PLI,
or ALL), a message is displayed and it continues executing as if COMP(ALL)
was specified.

Note: The ALL parameter for LEDATA also generates a report that includes all
the component control blocks.

ALL  Requests all above reports, as well as C/C++, COBOL, and PL/I reports.

Data selection parameters

Data selection parameters limit the scope of the data in the report. If no data
selection parameter is selected, the default is DETAIL.
DETail
Requests formatting all control blocks for the selected components. Only significant fields in each control block are formatted. For the Heap and Storage Management Reports, the DETAIL parameter will provide a detailed heap segment report for each heap segment in the dump. The detailed heap segment report includes information on the free storage tree in the heap segments, and all allocated storage elements. This report will also identify problems detected in the heap management data structures. For more information about the Heap Reports, see “Understanding the HEAP LEDATA output” on page 114.

EXCeption
Requests validating all control blocks for the selected components. Output is only produced naming the control block and its address for the first control block in a chain that is invalid. Validation consists of control block header verification at the very least.

For the Summary, CEEDUMP, C/C++, COBOL, and PL/I reports, the EXCEPTION parameter has not been implemented. For these reports, DETAIL output is always produced.

Control block selection parameters
Use these parameters to select the control blocks used as the starting points for formatting.

CAA(caa-address)
specifies the address of the CAA. If not specified, the CAA address is obtained from the TCB.

DSA(dsa-address)
specifies the address of the DSA. If not specified, the DSA address is assumed to be the register 13 value for the TCB.

TCB(tcb-address)
specifies the address of the TCB. If not specified, the TCB address of the current TCB from the CVT is used.

ASID(address-space-id)
specifies the hexadecimal address space ID. If not specified, the IPCS default address space ID is used. This parameter is not needed when the dump only has one address space.

NTHREADS(value)
specifies the number of TCBs for which the traceback will be displayed. If NTHREADS is not specified, value will default to (1). If value is specified as an asterisk (*), all TCBs will be displayed.

Examples
For examples of the output produced by LEDATA and explanation of the content, refer to “Understanding the Language Environment IPCS VERBEXIT LEDATA output.”

Understanding the Language Environment IPCS VERBEXIT LEDATA output
The Language Environment IPCS VERBEXIT LEDATA generates formatted output of the Language Environment runtime environment control blocks from a system dump. The following example illustrates the output produced when the LEDATA VERBEXIT is invoked with the ALL parameter. (Ellipses are used to summarize
some sections of the dump. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option when running the program CELSAMP in Figure 8 on page 48.

“Sections of the Language Environment LEDATA VERBEXIT formatted output” on page 109 describes the information contained in the formatted output. For reference, the sections of the sample dump are numbered to correspond with the descriptions of the formatted output.

**Figure 18. Example of formatted output from LEDATA VERBEXIT (Part 1 of 18)**

<table>
<thead>
<tr>
<th>Language Environment Product</th>
<th>04 V0I R0D.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] TCB: 00B86968</td>
<td>LE Level: 19</td>
</tr>
<tr>
<td>[2] Active Members: C/C++</td>
<td></td>
</tr>
<tr>
<td>[3] Language Environment DATA</td>
<td>Language Environment Product 04 V0I R0D.00</td>
</tr>
<tr>
<td>TCB: 00B86968</td>
<td>LE Level: 19</td>
</tr>
<tr>
<td>Active Members: C/C++</td>
<td></td>
</tr>
<tr>
<td>CEECA: 21616688</td>
<td>+000000 FLG0:00 LANGP:08 B05:21D71018 EOS:00000000</td>
</tr>
<tr>
<td>+000044 TORC:00000000 TOVF:80011410 ATTN:2160CF80</td>
<td></td>
</tr>
<tr>
<td>+001000 HILLEXIT:00000000 HOOK:55C000064 0DC50C00 CD60DCE</td>
<td></td>
</tr>
<tr>
<td>+0001A4 DIMA:000092A4 ALLOC:0700C3CB STATE:0700C3CB</td>
<td></td>
</tr>
<tr>
<td>+001800 ENTRY:0700C3CB EXIT:0700C3CB MEXIT:0700C3CB</td>
<td></td>
</tr>
<tr>
<td>+001BC LABEL:0700C3CB BCALL:0700C3CB ACALL:0700C3CB</td>
<td></td>
</tr>
<tr>
<td>+001C8 C:0700C3CB IFTRUE:0700C3CB IFFALSE:0700C3CB</td>
<td></td>
</tr>
<tr>
<td>+001D4 WHEN:0700C3CB OTHER:0700C3CB COT0:0700C3CB</td>
<td></td>
</tr>
<tr>
<td>+001F0 CGENE:216127EB CRENT:2160E148 CTHD:216109F0</td>
<td></td>
</tr>
<tr>
<td>+002100 EDCV:ABD931C CDB:21611A88 EDCV:21B00614</td>
<td></td>
</tr>
<tr>
<td>+0025B TCASRV_USERWORD:00000000 TCASRV_WORKAREA:2160C760</td>
<td></td>
</tr>
<tr>
<td>+002690 TCASRV_GETMAIN:00000000 TCASRV_FREEMAIN:00000000</td>
<td></td>
</tr>
<tr>
<td>+002680 TCASRV_LOAD:80000000 TCASRV_DELETE:80000000</td>
<td></td>
</tr>
<tr>
<td>+002790 TCASRV_GETMAIN:00000000 TCASRV_FREEMAIN:00000000</td>
<td></td>
</tr>
<tr>
<td>+002780 TCASRV_MESSAGE:00000000 WMS:00000000 SAVR:ABD9B2CA</td>
<td></td>
</tr>
<tr>
<td>+002800 SYST:03 HRDWR:03 SBSYS:02 FLG2:80 LEVEL:19</td>
<td></td>
</tr>
<tr>
<td>+0027F8 PM:04 GETS:80000000 CELV:A1618000 GETS:80000000</td>
<td></td>
</tr>
</tbody>
</table>
Figure 19. Example of formatted output from LEDATA VERBEXIT (Part 2 of 18)
Figure 20. Example of formatted output from LEDA VERBEXIT (Part 3 of 18)
<table>
<thead>
<tr>
<th>LAST WHERE</th>
<th>SET</th>
<th>Override</th>
<th>OPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>ABPERR(NONE)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>ABTERM(NCABEND)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>NOAIXBLD</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>ALL31(ON)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>ANYHEAP(0000016384,0000008192,ANY,FREE)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>NOAUTOTASK</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>BELOWHEAP(00008192,00004096,FREE)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>CBLOPTS(ON)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>CBLPSHPOP(ON)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>CBLOAD(FF)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>CEDUMP(0000000000,SYNSOUT=*,FREE-END,SPIN=UNALLOC)</td>
<td></td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>CHECK(ON)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>COUNTRY(US)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>DEPTHCONDLMT(0000000010)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>DYNQUMP(POSIX.DEBUGG,HLE7780,DYNAMIC,TMPG)</td>
<td></td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>ENVAR(&quot;**&quot;)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>ERRCOUNT(0000000000)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>ERUNUNIT(00000006)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>FILEHIST</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>FILETAG(NOAUTOCVT,N OAUTOTAG)</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>SETTING</td>
<td>OVR</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>HEAP_CHK(ON,0000000001,0000000000,0000000010,0000000010,0001024,0001024,00)</td>
<td></td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>HEAPCHK(ON,0000000001,0000000000,0000000010,0000000010,0001024,0001024,00)</td>
<td></td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>HEAPPOOLS(ON,00000008,00000010,00000032,00000010,00000128,00000010,00000256,00000010,00001024,00000010,00002048,00000010,00000000,00000010,00000000,00000010,00000000,00000010,00000000,00000010)</td>
<td></td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>HEAP(0000032768,0000032768,ANY,KEEP,000008192,000004096)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>INFO_MSGFILTER(OFF)</td>
<td></td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>INQPCOPN</td>
<td></td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>MSGFILE(SYSOUT,FBA,00000121,00000000,NOENQ)</td>
<td></td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>MSGQ(0000000000)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>IGNORED</td>
<td>OVR</td>
<td>NONONIPTSTACK(See THREADSTACK)</td>
<td></td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>OCSTATUS</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>PAGEFRAMESIZE(4K,4K,4K)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>PLSTACK(0000000000)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>PMAP(0000000020)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>POSIX(ON)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>PROFILE(OFF,&quot;**&quot;)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>PKUNUNIT(00000006)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>PUNUNIT(00000007)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>RDRUNIT(00000005)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>RECPAD(OFF)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>RTPROPT(ON)</td>
<td></td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>RTPTSTG(ON)</td>
<td></td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>NORTEREUS</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>NOSIMVRD</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>NOSIMWRD</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>STACK(00000131072,00000131072,ANY,KEEP,00000524288,00000131072)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>STORAGE(4A,8B,CC,000000000)</td>
<td></td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>OVR</td>
<td>TERMINOLOG(UIDUMP,CEE,0000000096)</td>
<td></td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>NOTEST(ALL,*,PROMPT,INSPPREF)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>THREADHEAP(0000004096,0000004096,FREE)</td>
</tr>
<tr>
<td>IBM-SUPPLIED</td>
<td>DEFAULT</td>
<td>OVR</td>
<td>THREADSTACK(OFF,0000004096,0000004096,ANY,KEEP,00000131072,00000131072)</td>
</tr>
</tbody>
</table>

Figure 21. Example of formatted output from LEDATA VERBEXIT (Part 4 of 18)
Heap Storage Control Blocks

Heappools trace available. To display: IP VERBX LEDATA 'HPT(*) CAA(21616BBB)'

ENSM: 21615CDB
+000000 EYE_CATCHER:ENSM ST_HEAP_ALLOC_FLAG:00000000
+000008 ST_HEAP_ALLOC_VAL:AA000000 ST_HEAP_FREE_FLAG:00000001
+000010 ST_HEAP_FREE_VAL:BB000000 REPORT_STORAGE:21615D04
+000018 UHEAP:CB73C7C2 21D91018 21D91018 00008000 00000000 00000200 00001000 00000000 00 000040 AHEAP:CB73C7C2 21D6D000 21D91018 00004000 00002000 00002000 00001000 00000000 00 000078 BHEAP:CB73C7C2 21615D50 21D91018 00000000 00000000 00000800 00000800 00000200 00001000 00000000 00 +0000A8 ENSM_ADDL_HEAP+S:22118238

STSB: 21615D04
+000000 EYE_CATCHER:STSB CHRP REQ:00000002 DSHP_REQ:00000001
+000008 IPT_INIT_SIZE:00020000 NONIPT_INIT_SIZE:00020000
+000014 IPT_INCR_SIZE:00020000 NONIPT_INCR_SIZE:00020000
+00001C THEAP_MAX_STOR:00000000

Enclave Level Stack Statistics

SKSB: 21615E2C
+000000 MAX_ALLOC:00009E90 CURR_ALLOC:00003EA0
+000008 LARGEST:00009E90 GETMAINS:00000001
+000010 FREEMAINS:00000000

SKSB: 21615E74
+000000 MAX_ALLOC:0000013C CURR_ALLOC:00000000
+000008 LARGEST:00000000 GETMAINS:00000002
+000010 FREEMAINS:00000000

SKSB: 21615E60
+000000 MAX_ALLOC:00000000 CURR_ALLOC:00000000
+000008 LARGEST:00000000 GETMAINS:00000001
+000010 FREEMAINS:00000000

User Heap Control Blocks

HPCB: 21615CFO
+000000 EYE_CATCHER:HPCB FIRST:21D91018 LAST:21D91018

HPSP: 21615D04
+000000 BYTES_ALLOC:00005248 CURR_ALLOC:00005248
+000008 GET_REQ:00000005 FREE_REQ:00000000
+000010 GETMAINS:00000001 FREEMAINS:00000000

HPSP: 21615EB8
+000000 BYTES_ALLOC:00000000 CURR_ALLOC:00000000
+000008 GET_REQ:00000000 FREE_REQ:00000000
+000010 GETMAINS:00000000 FREEMAINS:00000000

HANC: 21D91010
+000000 EYE_CATCHER:HANC NEXT:21615CFO PREV:21615CFO
+00000C HACPID:00000000 SEG_ADDR:A1D91018 ROOT_ADDR:21D96260
+000018 SEG_LEN:00000000 ROOT_LEN:00000000

This is the last heap segment in the current heap.

Figure 22. Example of formatted output from LEDATA VERBX (Part 5 of 18)
### Free Storage Tree for Heap Segment 21D91018

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21D96260</td>
<td>000020B8</td>
</tr>
</tbody>
</table>

Map of Heap Segment 21D91018

To display entire segment: IP LIST 21D91018 LEN(X'00008000') ASID(X'01A9')

### Map of Heap Segment 21D91018:

**Allocated storage elements**

1. **Node 21D91040**
   - Address: 21D96260
   - Length: 00002DB8
   - Parent: 0
   - Left: 0
   - Right: 0

**Summary of analysis for Heap Segment 21D91018:**

- **Amounts of identified storage:**
  - Free: 00002DB8
  - Allocated: 00005228
  - Total: 00007FE0

- **Number of identified areas:**
  - Free: 1
  - Allocated: 5
  - Total: 6

- No errors were found while processing this heap segment.

This is the last heap segment in the current heap.

### Anywhere Heap Control Blocks

- **HPCB:** 21615D20
  - EYE_CATCHER:HPCB FIRST: 21D6D000 LAST: 22177000
- **HPSB:** 21615E04
  - EYE_CATCHER:HPSB FIRST: 21615E00 LAST: 22177000
- **HANC:** 21D6D000
  - EYE_CATCHER:HANC FIRST: 21D6D000 LAST: 22177000

### Free Storage Tree for Heap Segment 21D90600

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21D90620</td>
<td>000000E0</td>
</tr>
</tbody>
</table>

**Summary of analysis for Heap Segment 21D91018:**

- **Amounts of identified storage:**
  - Free: 00002DB8
  - Allocated: 00005228
  - Total: 00007FE0

- **Number of identified areas:**
  - Free: 1
  - Allocated: 5
  - Total: 6

- No errors were found while processing this heap segment.

This is the last heap segment in the current heap.

**Figure 23. Example of formatted output from LEDATA VERBEXIT (Part 6 of 18)**
Map of Heap Segment 21D60000

To display entire segment: IP LIST 21D60000 LEN(X'000004000') ASID(X'01A9')

21D60020: Allocated storage element, length=00000001B. To display: IP LIST 21D60020 LEN(X'00000001B') ASID(X'01A9')

To display: IP LIST 21D60020 LEN(X'00000001B') ASID(X'01A9')

Below Heap Control Blocks

HPCB: 21D65050
+000000 EYE_CATCHER: HPBC FIRST:21D65050 LAST:21D65050
** NO SEGMENTS ALLOCATED **

HPSB: 21D65EC1
+000000 BYTES_ALLOC:00000000 CURR_ALLOC:00000000
+000000 GET_REQ:00000000 FREE_REQ:00000000
+000010 GETMAINS:00000000 FREEMAINS:00000000

Additional Heap Control Blocks

HPSB: 21D65E34
+000000 BYTES_ALLOC:00000000 CURR_ALLOC:00000000
+000000 GET_REQ:00000000 FREE_REQ:00000000
+000010 GETMAINS:00000000 FREEMAINS:00000000

ADHP: 22118238
+000000 EYE_CATCHER: ADHP NEXT:FOF00000 HEAPID:22118244

HPCB: 22118244
+000000 EYE_CATCHER: HPBC FIRST:22118240 LAST:22118240

HANC: 22118200
+000000 EYE_CATCHER: HANC NEXT:22118244 PREV:22118244
+000000 HEAPID:22118244 SEG_ADDR:22118200 ROOT_ADDR:22118200
+000010 SEG_LEN:00001000 ROOT_LEN:00000D38

This is the last heap segment in the current heap.

Free Storage Tree for Heap Segment 2212B000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Length</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2212B2C8</td>
<td>000000D3B</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 2212B000

To display entire segment: IP LIST 2212B0000 LEN(X'000001000') ASID(X'01A9')

2212B020: Allocated storage element, length=00000000. To display: IP LIST 2212B020 LEN(X'00000000') ASID(X'01A9')

2212B028: D7C3C2 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |PCIB......................... |

2212B2C8: Free storage element, length=00000000. To display: IP LIST 2212B2C8 LEN(X'00000000') ASID(X'01A9')

Summary of analysis for Heap Segment 2212B000:

Number of identified areas : Free: 1 Allocated: 1 Total: 2
00000000 bytes of storage were not accounted for.
No errors were found while processing this heap segment.
This is the last heap segment in the current heap.

Figure 24. Example of formatted output from LEDA VERBEXIT (Part 7 of 18)
Heap Pool Report

```
OPCB: 21D96000
+000000 EYE_CATCHER:OPCB LENGTH:00000008 NUMPOOLS:0000001
+000000 LARGEST_CELL_SIZE:00000000 BIG_REQUESTS:00000000
+000014 STORAGE_HITS_ADDR:21F00028 FLAGS:100 NUMGETARRAYS:00
+000018 NUMCELLSIZES:06 GET_POOLINFO_ARRAYS_PTR:21D96A28
```

Data for pool 1:
```
POOLDATA: 21D96000
+000000 POOL_INDEX:00000001 INPUT_CELL_SIZE:00000008
+000008 CELL_SIZE:000000010 INPUT_PERCENT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000008 POOL_LATCH_ADDR:21E44B04 POOL_INDEX:00000000
+000029 LAST_CELL:21D95620 NEXT_CELL:21D95620
+000008 Q_CONTROL_INFO:00000000 Q_First_Cell:00000000
+000038 POOL_NUM_GET_TOTAL:00000000 POOL_NUM_FREE:00000000
+000000 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SAME_SIZE:01
+000000 POOL_INDEX_SIZE:01 POOL_INDEX_SIZE:01
+000000 POOL_TRACE_TABLE:21E4F050
```

Heap Pool Extent Mapping
```
EXTENT: 21D95598
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire pool extent: IP LIST 21D95598 LEN(X'000000CC') ASID(X'01A9')
21D955A0: Allocated storage cell. To display: IP LIST 21D955A0 LEN(X'00000010') ASID(X'01A9')
21D955A8: 21D955B0 AAAA AAAA 00000000 00000000|.
21D955B0: Allocated storage cell. To display: IP LIST 21D955B0 LEN(X'00000010') ASID(X'01A9')
21D955BA: 21D955C0 AAAA AAAA 00000000 00000000|.
Summary of analysis for Pool 1:
Number of cells: Unused: 202 Free: 0 Allocated: 2 Total Used: 204
00000000 free cells were not accounted for.
No errors were found while processing this Pool.
```

Data for pool 2:
```
POOLDATA: 21D960C0
+000000 POOL_INDEX:00000002 INPUT_CELL_SIZE:00000020
+000008 CELL_SIZE:00000020 INPUT_PERCENT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000008 POOL_LATCH_ADDR:21E44B08 POOL_INDEX:00000000
+000029 LAST_CELL:00000000 NEXT_CELL:00000000
+000008 Q_CONTROL_INFO:00000000 Q_First_Cell:00000000
+000038 POOL_NUM_GET_TOTAL:00000000 POOL_NUM_FREE:00000000
+000000 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SAME_SIZE:01
+000000 POOL_INDEX_SIZE:02 POOL_INDEX_SIZE:02
+000000 POOL_TRACE_TABLE:21EDF070
```

There are no extents for this pool.
```
```

Data for pool 6:
```
POOLDATA: 21D96E00
+000000 POOL_INDEX:00000006 INPUT_CELL_SIZE:00000080
+000008 CELL_SIZE:00000080 INPUT_PERCENT:00000000
+000010 CELL_POOL_SIZE:00000020 CELL_POOL_NUM:00000000
+000018 POOL_LATCH_ADDR:21E44C38 POOL_INDEX:00000000
+000029 LAST_CELL:21D94D78 NEXT_CELL:21D93758
+000028 Q_CONTROL_INFO:00000000 Q_First_Cell:00000000
+000038 POOL_NUM_GET_TOTAL:00000001 POOL_NUM_FREE:00000000
+000000 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SAME_SIZE:01
+000000 POOL_INDEX_SIZE:06 POOL_INDEX_SIZE:06
+000000 POOL_TRACE_TABLE:21F9F0F0
```

Heap Pool Extent Mapping
```
EXTENT: 21D93568
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire pool extent: IP LIST 21D93568 LEN(X'00000028') ASID(X'01A9')
21D93570: Allocated storage cell. To display: IP LIST 21D93570 LEN(X'00000008') ASID(X'01A9')
21D93578: 00000000 21D703CC 00000000 00000000|.
Summary of analysis for Pool 6:
Number of cells: Unused: 3 Free: 0 Allocated: 1 Total Used: 4
00000000 free cells were not accounted for.
No errors were found while processing this Pool.
```

Figure 25. Example of formatted output from LEDATA VERBEXIT (Part 8 of 18)
Figure 26. Example of formatted output from LEDATA VERBEXIT (Part 11 of 18)
To display entire DSA: IP LIST 21D71020 LEN(X'000030F8') ASID(X'01A9')

DSA: 21D7124B
+000000 FLAGS:10CC MEMD:CCCC RKC:21D71130 FW:21D71328
+000000 R1:21600A66 R15:21600A45 R0:216011A4
+000000 R2:21D7112E R2:21D71305 R3:216000FA
+000000 R4:21D71102 R5:21600ED0 R6:21D7113C
+000000 R7:21D71310 R8:00000030 R9:8B000000
+000000 LWS:00000000 NAB:21D71328 PNAB:CCCCCCCC
+000064 RENT:CCCCCCCC CILE:CCCCCCCC MODM:CCCCCCCC
+000078 RMR:CCCCCCCC

Contents of DSA at location 21D7124B:

+00000000 10CCCCCC 21D71130 21D71328 A1600666 2180445C 216011A4 21D7120E 21D71305 ...P...P...w...u...P...
+00000020 216000FA 21D71302 21600E0D 21D7130C 21D71310 00000030 00000000 A02E8822 ...P...P...P...P...P...KY.
+00000040 A169F4B8 21616888 00000000 21D712E6 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC /...P...
+00000060 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC 216011A4 21601195 ...
+00000080 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC 00000000 ...
+000000A0 21600E0C 22128948 00000000 22081178 22082388 00000000 00000000 216614A8 CCCCCCCC ...
+000000C0 00000000 00000000 22128E40 00000000 00000000 00000000 216614A8 CCCCCCCC ...

DSA: 21D71130
+000000 FLAGS:10CC MEMD:CCCC RKC:21D71030 FW:CCCCCCCC
+000000 R14:A12D8EB0 R15:A160000C R8:21D7124B
+000001 R1:21D6E1E8 R2:A1D2E8B0 R3:00000002
+000004 R4:A169303A R5:21615700 R6:216039EC
+000008 R7:21604330 R8:00000030 R9:8B000000
+00000C R10:A1D2E8B8 R11:A1692F48 R12:21616888
+00000E LWS:00000000 NAB:21D7124B PNAB:CCCCCCCC
+00000E RENT:CCCCCCC CILE:CCCCCCC MODM:CCCCCCCC
+000078 RMR:CCCCCCCC

Contents of DSA at location 21D71130:

+00000000 10CCCCCC 21D71030 CCCCCCCC A2182E0B A160000C 21D7124B 216E1E8B A1D2EB080 ...
+00000020 00000002 A169303A 21615700 216039EC 21604330 00000030 00000000 A1D2EB080 ...
+00000040 A169F4B8 21616888 00000000 21D712E6 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC...
+00000060 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC...
+00000080 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC...
+000000A0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC...
+000000C0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC...

To display entire DSA: IP LIST 21D71130 LEN(X'0000001B8') ASID(X'01A9')

DSA: 21D71030
+000000 FLAGS:0000 MEMD:CCCC RKC:21617660 FW:CCCCCCCC
+000000 R14:A169310E R15:21D2EB8E R0:7D000009
+000001 R1:21D71080 R2:21604328 R3:00000002
+000004 R4:A169303A R5:21615700 R6:216039EC
+000008 R7:21604330 R8:00000030 R9:8B000000
+00000C R10:A1D2EBB2 R11:A1692F48 R12:21616888
+00000E LWS:00000000 NAB:21D71310 PNAB:CCCCCCCC
+000064 RENT:CCCCCCC CILE:CCCCCCC MODM:CCCCCCCC
+000078 RMR:CCCCCCC

Contents of DSA at location 21D71030:

+00000000 00000000 21617660 CCCCCCCC A169310E 21D2EB8E 7D000009 21D71080 2160432B ...
+00000020 00000002 A169303A 21615700 216039EC 21604330 00000030 00000006 A1D2EB8E ...
+00000040 A169F4B8 21616888 00000000 21D71130 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC ...
+00000060 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC ...
+00000080 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC ...
+000000A0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC ...
+000000C0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC ...

User Stack Control Blocks

STK#: 21D71018
+000000 EYE_CATCHER:STKU NEXT:2161742C PREV:2161742C
+000000 SEGMENT_LEN:00002000

Figure 27. Example of formatted output from LEDATA VERBEXIT (Part 12 of 18)
Library Stack Control Blocks

STKH: 00015000
+000000 EYE_CATCHER;STKL NEXT:21617470  PREV:21617470
+00000C SEGMENT_LEN:00001000

[12] Condition Management Control Blocks

HCOM: 2160CF38
+000000 PICA_AREA:00000000 00000000 EYES:HCOM CA_APTR1:21616BB8
+000014 CTVD6:98 FLAG:60F0C000 EXIT_STK:2121E8E8
+000020 RSM_PTR:00000000  HDLL_STK:2121E8E8
+000028 SRP_TOKEN:00000000  CSTK:21330E48  CIBH:21D72B28
+0000B4 COND_LOG:21617470  DSA_4083:00000000
+0000C0 SHUNT_ADDR:216FB7CA  A16FB7CA
+0000CC SHUNT_REG0:00000001  SHUNT_REG1:21D77050
+000800 SHUNT_REG2:00000000  SHUNT_REG3:00000000
+000808 SHUNT_REG4:00000000  SHUNT_REG5:00000000
+000810 SHUNT_REG6:00000000  SHUNT_REG7:00000000
+000818 SHUNT_REG8:00000000  SHUNT_REG9:216FB7CA
+00081C SHUNT_REG10:00000000  SHUNT_REG11:A16FA5B
+000820 SHUNT_REG12:21616BB8  SHUNT_REG13:00000000
+000828 SHUNT_REG14:A16FB6D6  SHUNT_REG15:00000000
+000830 SHUNT_CODE1:00000000  SHUNT_CODE2:00000004
+000838 SHUNT_CODE3:00000000  SHUNT_CODE4:00000000
+00083C SHUNT_CODE5:00000000  SHUNT_CODE6:00000000
+000840 SHUNT_CODE7:00000000  SHUNT_CODE8:00000000
+000844 SHUNT_CODE9:00000000  SHUNT_CODE10:00000000
+000848 SHUNT_CODE11:00000000  SHUNT_CODE12:00000000
+00084C SHUNT_CODE13:00000000  SHUNT_CODE14:00000000
+000850 SHUNT_CODE15:00000000

CIBH: 21D72B28
+000000 EYE:CIBH BACK:2160E430  FRWD:00000000
+000010 PTR_CIB:00000000  FLAG1:00  ERROR_LOCATION_FLAGS:00
+000020 HDLQ:00000000  STATE:00000000  PRM_DESC:00000000
+000034 PRM_PREFIX:00000000
+000054 CIB_SIZ:0000  CIB_VER:0000  FLG_1:00  FLG_2:00  FLG_3:00
+00005E FLG_4:00  FLG_5:00  FLG_6:00  FLG_7:00  FLG_8:00  ABCD:00000000
+000070 OLD_COND_64:00000000  OLD_MIB:00000000  COND_64:00000000
+000080 SV1:00000000  SV2:00000000  INT:00000000  MID:00000000
+000090 HDL_SF:00000000  HDL_EPT:00000000  HDL_RST:00000000
+0000A8 COND_DEFAULT:00000000  Q_DATA_TOKEN:00000000  FDBK:00000000
+0000B8 ABNAME:........ BBRANCH_OFFSET:00000000
+000220 BBRANCH_STMTID:........ BBRANCH_STMTLEN:0000

Machine State
+000248 MCH_EYE:....
+000250 GPR00:00000000  GPR01:00000000
+000258 GPR02:00000000  GPR03:00000000
+000260 GPR04:00000000  GPR05:00000000
+000268 GPR06:00000000  GPR07:00000000
+000270 GPR08:00000000  GPR09:00000000
+000278 GPR10:00000000  GPR11:00000000
+000280 GPR12:00000000  GPR13:00000000
+000288 GPR14:00000000  GPR15:00000000
+000290 PSW:00000000  00000000
+000298 ILC:00000000  IC1:00  IC2:00  PFT:00000000
+0002A0 FLT_0:00000000  00000000  FLT_2:00000000  00000000
+0002B0 FLT_4:00000000  00000000  FLT_6:00000000  00000000
+0002C0 INT_SF:00000000  FLAGS:00  EXT:00000000  BEA:00000000
+000308 SAVSTACK_ASYNC_PTR:00000000
+000318 FLT_1:00000000  00000000  
+000320 FLT_3:00000000  00000000  
+000328 FLT_5:00000000  00000000  
+000330 FLT_7:00000000  00000000  
+000338 FLT_8:00000000  00000000  FLT_9:00000000  00000000
+000348 FLT_10:00000000  00000000  FLT_11:00000000  00000000
+000358 FLT_12:00000000  00000000  FLT_13:00000000  00000000
+000368 FLT_14:00000000  00000000  FLT_15:00000000  00000000

Figure 28. Example of formatted output from LEDA VERBEXIT (Part 13 of 18)
Figure 29. Example of formatted output from LEDATA VERBEXIT (Part 14 of 18)
Figure 30. Example of formatted output from LEDATA VERBEXIT (Part 15 of 18)

CMXB: 216151A0
+000000 EYE:CMXB SIZE:0148 FLAGS:8000 DHEAD1:00016000
+000000 DHEAD2:00012000

MDST forward chain from CMXBDHEAD(1)

MDST: 00016000
+000000 EYE:MDST SIZE:0100 CTL:40 CEDUMPLOC:00
+000008 NEXT:00012000 PREV:00016000 DNAME:CEEDUMP

MDST: 00012000
+000000 EYE:MDST SIZE:0100 CTL:40 CEDUMPLOC:00
+000008 NEXT:00000000 PREV:00016000 DNAME:SYSOUT

MDST: 00016000
+000000 EYE:MDST SIZE:0100 CTL:40 CEDUMPLOC:00
+000008 NEXT:00012000 PREV:00000000 DNAME:CEEDUMP

TMXB: 2160F048
+000000 EYE:TMXB MIB_CHAIN_PTR:22167028

MGF: 22167028
+000000 EYE:CMIB PREV:22113780 NEXT:22113880 SEQ:00000005
+000010 CTOK:00000BFF 41C3C5C5 (CEE3063I)

MGF: 22113780
+000000 EYE:CMIB PREV:22167028 NEXT:2160F080 SEQ:00000002
+000010 CTOK:00000BCC 59C3C5C5 (CEE3209S)

MGF: 2160F080
+000000 EYE:CMIB PREV:22113800 NEXT:2211315C0 SEQ:00000001
+000010 CTOK:00000B06 41C3C5C5 (CEE3574I)

MGF: 2211315C0
+000000 EYE:CMIB PREV:2160F080 NEXT:22113780 SEQ:00000003
+000010 CTOK:00000B3E 59C3C5C5 (CEE304625)

MGF: 22113780
+000000 EYE:CMIB PREV:2211315C0 NEXT:22167028 SEQ:00000004
+000010 CTOK:00000B37 49C3C5C5 (CEE3455W)

[14] Information for enclave main

[15] Information for thread 27ACD200000000000
PCB Address: 21615320
TCB Address: 008E6968

[16] Registers and PSW:
GPR0..... 00000000_84000000 GPR1..... 00000000_84000F7C GPR2..... 00000000_21D76218 GPR3..... 00000000_00020009
GPR4..... 00000000_21615320 GPR5..... 00000000_2160E430 GPR6..... 00000000_21D72618 GPR7..... 00000000_21D72C32
GPR8..... 00000000_21D72618 GPR9..... 00000000_21D721AC GPR10..... 00000000_21D72D1F GPR11..... 00000000_2169F8B
GPR12..... 00000000_21616B88 GPR13..... 00000000_21D74E18 GPR14..... 00000000_A16D9B58 GPR15..... 00000000_00000000
PSW..... 07B01400 A169C32

Figure 31. Example of formatted output from LEDATA VERBEXIT (Part 16 of 18)
Figure 32. Example of formatted output from LEDA VERBEXIT (Part 17 of 18)
Sections of the Language Environment LEDATA VERBEXIT formatted output
The sections of the output listed in Table 21 appear independently of the Language Environment-conforming languages used.

Table 21. Contents of the LEDATA VERBEXIT formatted output

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] - [9] Summary:</td>
<td>The following sections are included when the SUMMARY parameter is specified on the LEDATA invocation.</td>
</tr>
</tbody>
</table>

Chapter 3. Using Language Environment debugging facilities

Figure 33. Example of formatted output from LEDATA VERBEXIT (Part 18 of 18)
<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [1] Summary Header        | Contains the following information:  
|                           | • Address of Thread control block (TCB)  
|                           | • Release number  
|                           | • Address Space ID (ASID) |
| [2] Active Members List | List of active members is extracted from the enclave member list (MEML) |
| [4] CEEDLLF | Formats the contents of all Language Environment CEEDLLF (DLLF) control blocks that are in use. See CEEDLLF — DLL failure control block in z/OS Language Environment Vendor Interfaces for more information about the CEEDLLF control block chain. |
| [5] CEEPCB | Formats the contents of the Language Environment process control block (PCB), and the process level member list. |
| [7] CEEEDB | Formats the contents of the Language Environment enclave data block (EDB), and the enclave level member list. |
| [8] PMCB | Formats the contents of the Language Environment program management control block (PMCB). |
| [9] Runtime Options | Lists the runtime options in effect at the time of the dump, and indicates where they were set. |
| [10] Heap Storage Control Blocks | This section is included when the HEAP or SM parameter is specified on the LEDATA invocation. It formats the Enclave-level storage management control block (ENSM) and for each different type of heap storage:  
|                           | • Heap control block (HPCB)  
|                           | • Chain of heap anchor blocks (HANC). A HANC immediately precedes each segment of heap storage.  
|                           | This section includes a detailed heap segment report for each segment in the dump. For more information about the detailed heap segment report, see “Understanding the HEAP LEDATA output” on page 114.  
|                           | When HEAPPOOLS is ON, this section also includes a detailed heap pools report. For more information about the detailed heap pools report, see “Understanding the heap pools LEDATA output” on page 119. |
| [11] Stack Storage Control Blocks | This section is included when the STACK or SM parameter is specified on the LEDATA invocation; it formats:  
|                           | • Storage management control block (SMCB)  
|                           | • Chain of dynamic save areas (DSA). See “Upward-growing (non-XPLINK) stack frame section” on page 66 or “Downward-growing (XPLINK) stack frame section” on page 67 for a description of the fields in the DSA.  
|                           | • Chain of stack segment headers (STKH). An STKH immediately precedes each segment of stack storage. |
| [12] Condition Management Control Blocks | This section is included when the CM parameter is specified on the LEDATA invocation; it formats the chain of Condition Information Block Headers (CIBH) and Condition Information Blocks. The Machine State Information Block is contained with the CIBH starting with the field labeled MCH_EYE. See “Condition information block” on page 78 for a description of fields in these control blocks. |
| [13] Message Processing Control Blocks | This section is included when the MH parameter is specified on the LEDATA invocation. |
### Table 21. Contents of the LEDA VERBEXIT formatted output (continued)

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[14]-[17] NTHREADS</td>
<td>One or more instances of these sections are included when the NTHREADS() parameter is specified on the LEDATA invocation. For a description of NTHREADS, see “Report type parameters” on page 90.</td>
</tr>
<tr>
<td>[14] - [21] CEEDUMP</td>
<td>These sections are included when the CEEDUMP parameter is specified on the LEDATA invocation.</td>
</tr>
<tr>
<td>[14] Enclave Identifier</td>
<td>Names the enclave for which information is provided.</td>
</tr>
<tr>
<td>[15] Information for thread</td>
<td>Shows the system identifier for the thread. Each thread has a unique identifier.</td>
</tr>
<tr>
<td>[16] Registers and PSW</td>
<td>Displays the register and program status word (PSW) values that were used to create the traceback. These values may come from the TCB, the RTM2 work area, a linkage stack entry or output from the BPXGMSTA service. This section is not displayed when the DSA() parameter is specified on the LEDATA invocation.</td>
</tr>
</tbody>
</table>
[17] Traceback

For all active routines in a particular thread, the traceback section shows routine information in two parts. The first part contains the following items:

- **DSA number**: A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second part of the traceback.
- **Entry**: For COBOL, Fortran, and PL/I routines, this is the entry point name. For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, the string "** NoName **" will appear.
- **Entry point offset**
- **Statement number**: This field contains no Language Environment data.
- **Load module**
- **Program unit**: The primary entry point of the external procedure. For COBOL programs, this is the PROGRAM-ID name. For C, Fortran, and PL/I routines, this is the compile unit name. For Language Environment-conforming assemblers, this is the EPNAME = value on the CEEPPA macro.
- **Service level**: The latest service level applied to the compile unit (for example, for IBM products, it would be the PTF number).
  - If the service level string is equal or less than 7 bytes, all of the string will be output.
  - If the service level string is longer than 7 bytes, the Service column will only show the first 7 bytes of the service string, and the full service string will be shown in section of Full Service Level with max length of 64 bytes.
- **Status**: Routine status can be call, exception, or running.

The second part contains the following items:

- **DSA number**
- **Entry**
- **Service**: The full service level string with max length of 64 bytes will be displayed here.

The third part of the traceback, which is also referred to as the "Full Service Level" section, contains the following:

- **DSA number**
- **Entry**
- **Service**: The full service level string with max length of 64 bytes will be displayed here.
PTBL LEDATA output: The VERBEXIT LEDATA command generates formatted output of Preinit tables when the PTBL or ALL parameter are specified. If ALL is specified, PTBL defaults to CURRENT value. The following sample illustrates the output produced when the VERBEXIT LEDATA command is invoked with the PTBL parameter.

```
PTBL(CURRENT)
*******************************************************************************
PTBL(CURRENT) Output:
*******************************************************************************
 PTBL (CURRENT)
*******************************************************************************
 PTBL Environment Program Unit Trace Data
 System Trace Level: 0
 CEEPIPI Environment Trace Entry:
 CEEPIPI Environment Table Entry and Trace Entry:
 Active CEEPIPI Environment (Address 20905CB0)
 Eye catcher: CEEXIPTB
 TCB address: 00086E08
 CEEPIPI Environment:
 Non-XPLINK Environment
 Environment Type: MAIN
 Sequence of Calls not active
 Exits not established
 Signal Interrupt Routines not registered
 Service Routines not active
 CEEPIPI Environment Enclave Initialized
 Number of CEEPIPI Table Entries = 3

 CEEPIPI Table Entry Information:
 CEEPIPI Table Index 0 (Entry 1)
 Routine Name = ISJPPCA3
 Routine Type = C/C++
 Routine Entry Point = A0910530
 Routine Function Pointer = A0910620
 Routine Entry Is Non-XPLINK
 Routine was loaded by Language Environment
 Routine Address was resolved
 Routine Function Descriptor was valid
 Routine Return Code = 0
 Routine Reason Code = 0
 Entry of routine in CEEPIPI Table for Index 0 (209050B8)
 +000000 209050B8 A0910620 20919830 80000000 00000000 00000000 00000000 00000000 |j...j......................|
 +000020 209050D8 00000000 00000000 00000000 A0910530 00000000 20919708 00000003 20910530 |................JpE...J...|
 +000040 209050F8 00000000 00000000 C9E2D1D7 D713C1F3 00000000 00000000 00000008 00000000 |j..........................|

 CEEPIPI Table Index 1 (Entry 2) not in use.
```
Exiting Language Environment Data

When nested CEEPIPI main-DP environments are present, two new items will appear after the TCB address:

- Address of the CEEPIPI environment (PTBL) that called the currently displayed CEEPIPI environment.
- Saved register 13 value. This is the address of the DSA for the Language Environment routine called from the assembler CEEPIPI driver.

The following is an example:

Eyecatcher: CEEXPITB
TCB address: xxxxxxxx
Caller PTBL: xxxxxxxx
Saved R13: xxxxxxxx

Understanding the HEAP LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap segment report when the HEAP option is used with the DETAIL option, or when the SM,DETAIL option is specified. The detailed heap segment report is useful when trying to pinpoint damage because it provides very specific information. The report describes the nature of the damage, and specifies where the actual damage occurred. The report can also be used to diagnose storage leaks, and to identify heap fragmentation. The following example illustrates the output produced by specifying the HEAP option.

"Heap report sections of the LEDATA output" on page 118
**Page 118** describes the information contained in the formatted output. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows in **Table 22 on page 118**. Ellipses are used to summarize some sections of the dump.

**Note:** Language Environment does not provide support for alternative Vendor Heap Manager (VHM) data. LEDATA verb exit will state that an alternative VHM is in use.

```
IP VERBEXIT LEDATA 'HEAP'
******************************************************************************
LANGUAGE ENVIRONMENT DATA
******************************************************************************
Language Environment Product: 04 V01 R02.00

Heap Storage Control Blocks
  ENSM: 00014D30
       +0000AB ENSM_ADDL_HEAPS:259B1120

User Heap Control Blocks
  HPCB: 00014D48
       +000000 EYE_CATCHER:HPCB FIRST:25995000 LAST:25995000
  HANC: 25995000
       +000000 EYE_CATCHER:HANC NEXT:00014D48 PREV:00014D48
       +00000C HEAPID:00000000 SEG_ADDR:25995000 ROOT_ADDR:25995080
       +000018 SEG_LEN:00008000 ROOT_LEN:00007F50

This is the last heap segment in the current heap.

[1] Free Storage Tree for Heap Segment 25995000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Node Address</th>
<th>Node Length</th>
<th>Parent Address</th>
<th>Left Node Address</th>
<th>Right Node Address</th>
<th>Left Length</th>
<th>Right Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25995080</td>
<td>00007F50</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>


To display entire segment: IP LIST 25995000 LEN(X'00000000') ASID(X'0021')

25995020: Allocated storage element, length=00000038. To display: IP LIST 25995020 LEN(X'00000038') ASID(X'0021')
25995028: C3C43030 00000000 40000000 00000000 24700F90 24703F70 25993870 00000490 [CDLL.................]  
25995058: Allocated storage element, length=00000038. To display: IP LIST 25995058 LEN(X'00000038') ASID(X'0021')
25995060: C3C43030 25995028 00000000 00000000 24700F90 24703F70 25993870 00000490 [CDLL....0........]  
25995090: Allocated storage element, length=00000010. To display: IP LIST 25995090 LEN(X'00000010') ASID(X'0021')
25995098: 2590DB88 00000000
259950A0: Allocated storage element, length=00000010. To display: IP LIST 259950A0 LEN(X'00000010') ASID(X'0021')
259950A8: 2590DBE0 00000000
259950B0: Free storage element, length=00007F50. To display: IP LIST 259950B0 LEN(X'000007F50') ASID(X'0021')

Summary of analysis for Heap Segment 25995000:

Amounts of identified storage: Free:00007F50 Allocated:00000090 Total:00007FE0
Number of identified areas: Free: 1 Allocated: 4 Total: 5
00000000 bytes of storage were not accounted for.
No errors were found while processing this heap segment.
This is the last heap segment in the current heap.

Anywhere Heap Control Blocks

  HPCB: 00014D78
       +000000 EYE_CATCHER:HPCB FIRST:24A91000 LAST:259C2000
  HANC: 24A91000
       +000000 EYE_CATCHER:HANC NEXT:25993000 PREV:00014D78
       +00000C HEAPID:00014D78 SEG_ADDR:24A91000 ROOT_ADDR:00000000
       +000018 SEG_LEN:00F00000 ROOT_LEN:00000000

Chapter 3. Using Language Environment debugging facilities 115
Free Storage Tree for Heap Segment 24A91000

The free storage tree is empty.

Map of Heap Segment 24A91000

To display entire segment: IP LIST 24A91000 LEN(X'00F00028') ASID(X'0021')

24A91020: Allocated storage element, length=00F00008. To display: IP LIST 24A91020 LEN(X'00F00008') ASID(X'0021')
24A91028: B035F0DB B2C00081 2A40ED80 00000000 03000000 00000001 94B1B995 40404040 |EQ...ADO ........................main |

Summary of analysis for Heap Segment 24A91000:
Amounts of identified storage: Free:00000000 Allocated:00F00008 Total:00F00008
Number of identified areas = Free: 0 Allocated: 1 Total: 1
00000000 bytes of storage were not accounted for.
No errors were found while processing this heap segment.

| HANC: 259AC000 |
| +00000000 EYE_CATCHER:HANC NEXT:259AF000 PREV:2599D000 |
| +00000C HEAPID:00010410 SEG_ADDR:259AC000 ROOT_ADDR:259AC020 |
| +000018 SEG_LEN:00002000 |

Free Storage Tree for Heap Segment 259AC000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Node</th>
<th>Node</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>259AC020</td>
<td>00000C30</td>
<td>00000000</td>
<td>00000000</td>
<td>259AC040</td>
<td>00000000</td>
<td>00000038</td>
</tr>
<tr>
<td>1</td>
<td>259AC048</td>
<td>00000038</td>
<td>259AC020</td>
<td>00000000</td>
<td>259AC060</td>
<td>00000000</td>
<td>00000038</td>
</tr>
</tbody>
</table>

Map of Heap Segment 259AC000

To display entire segment: IP LIST 259AC000 LEN(X'00002000') ASID(X'0021')

259AC020: Free storage element, length=00000C30. To display: IP LIST 259AC020 LEN(X'00000C30') ASID(X'0021')

259AC0C0: Allocated storage element, length=00000728. To display: IP LIST 259AC0C0 LEN(X'00000728') ASID(X'0021')
259AC0C8: D033E340 01C0001 00000000 00000000 00000003 00010003 |LLT ..............................|
259AC0C8: Allocated storage element, length=00000728. To display: IP LIST 259AC0C8 LEN(X'00000728') ASID(X'0021')
259AC0C8: C033E340 01C0001 00000000 00000000 00000003 00010003 |LLT ..............................|

Free Storage Tree for Heap Segment 259AC0C00

<table>
<thead>
<tr>
<th>Depth</th>
<th>Node</th>
<th>Node</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>259AC020</td>
<td>00000C30</td>
<td>00000000</td>
<td>00000000</td>
<td>259AC040</td>
<td>00000000</td>
<td>00000038</td>
</tr>
<tr>
<td>1</td>
<td>259AC048</td>
<td>00000038</td>
<td>259AC020</td>
<td>00000000</td>
<td>259AC060</td>
<td>00000000</td>
<td>00000038</td>
</tr>
</tbody>
</table>

Map of Heap Segment 259AC0C00

To display entire segment: IP LIST 259AC0C00 LEN(X'00002000') ASID(X'0021')

259AC0C0: Free storage element, length=00000728. To display: IP LIST 259AC0C0 LEN(X'00000728') ASID(X'0021')

Summary of analysis for Heap Segment 259AC0C00:
Amounts of identified storage: Free:000000FF8 Allocated:00000FF8 Total:00001FF0
Number of identified areas = Free: 2 Allocated: 8 Total: 10
00000000 bytes of storage were not accounted for.
No errors were found while processing this heap segment.

Below Heap Control Blocks

HPCB: 00010408 +000000 EYE_CATCHER:HPCB FIRST:00044000 LAST:00044000

HANC: 00044000 +000000 EYE_CATCHER:HANC NEXT:00010408 PREV:00010408
+00000000 EYE_CATCHER:HANC NEXT:00010408 PREV:00010408
+00000C HEAPID:00010410 SEG_ADDR:00044000 ROOT_ADDR:00044388
+000018 SEG_LEN:00002000 ROOT_LEN:00001C78
This is the last heap segment in the current heap.

Free Storage Tree for Heap Segment 00044000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Node Address</th>
<th>Node Length</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00044388</td>
<td>0001C78</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 00044000

To display entire segment: IP LIST 00044000 LEN('000002000') ASID('00021')

00044020: Allocated storage element, length=00000048. To display: IP LIST 00044020 LEN('000000048') ASID('00021')

00044020: C8C4D3E2 00000000 00044220 00000040 00010000 00000001 000241E0 24701038 [HDLS.........................]

0004406B: Allocated storage element, length=00000128. To display: IP LIST 0004406B LEN('00000128') ASID('00021')

00044070: 07000700 05E0900F EA06A1DE 00258BC0 E11298DF E116980F E2C667D4 01200001 ..........w..........0....SFNM....

00044190: Allocated storage element, length=00000088. To display: IP LIST 00044190 LEN('00000088') ASID('00021')

00044190: C32E3D34 00000000 00030001 00000001 00000006 E0000000 00000000 00000000 [CSMK.........................]

00044218: Allocated storage element, length=00000128. To display: IP LIST 00044218 LEN('00000128') ASID('00021')

00044220: C8C4D3E2 00044208 00000000 00000010 00000000 00000002 000241E0 259A0890 [HDLS.. .................]

00044260: Allocated storage element, length=00000128. To display: IP LIST 00044260 LEN('00000128') ASID('00021')

00044260: 07000700 05E0900F EA06A1DE 00258BC0 E11298DF E116980F E2C667D4 01200001 ..........w..........0....SFNM....

00044388: Free storage element, length=0001C78. To display: IP LIST 00044388 LEN('000001C78') ASID('00021')

Summary of analysis for Heap Segment 00044000:

Amounts of identified areas: Free: 00001C78 Allocated: 00000036 Total: 00001FE0

Number of identified areas: Free: 1 Allocated: 5 Total: 6

00000000 bytes of storage were not accounted for.

No errors were found while processing this heap segment.

This is the last heap segment in the current heap.

Additional Heap Control Blocks

ADHP: 259B1120
  +000000 EYE_CATCHER:ADHP NEXT: 259B24A8 HEAPID: 259B112C

HPCB: 259B112C
  +000000 EYE_CATCHER:hpcb FIRST: 259B112C LAST: 259B112C

ADHP: 259B24A8
  +000000 EYE_CATCHER:ADHP NEXT: 259A0DC8 HEAPID: 259B24B4

HPCB: 259B24B4
  +000000 EYE_CATCHER:hpcb FIRST: 259B24B4 LAST: 259B24B4

ADHP: 259A0DC8
  +000000 EYE_CATCHER:ADHP NEXT: F0F00000 HEAPID: 259A0DC14

HPCB: 259A0DC14
  +000000 EYE_CATCHER:HPCB FIRST: 259AE000 LAST: 259AE000

HANC: 259AE000
  +000000 EYE_CATCHER:HANC NEXT: 259A0DC14 PREV: 259A0DC14
  +000000 HEAPID: 259A0DC14 SEG_ADDR: 259AE000 ROOT_ADDR: 259AE1EB
  +000018 SEG_LEN: 000001000 ROOT_LEN: 000000E18

This is the last heap segment in the current heap.

Free Storage Tree for Heap Segment 259AE000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Node Address</th>
<th>Node Length</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>259AE1EB</td>
<td>0000E1E0</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 259AE000

To display entire segment: IP LIST 259AE000 LEN('000001000') ASID('00021')

259AE020: Allocated storage element, length=0000001C8. To display: IP LIST 259AE020 LEN('0000001C8') ASID('00021')

259AE020: D7C3C9C2 00000000 00000000 00010018 00000000 00000000 00000000 00000000 00000000 00000000 [PCB:.........................]

259AE1EB: Free storage element, length=00000E18. To display: IP LIST 259AE1EB LEN('000000E18') ASID('00021')
Summary of analysis for Heap Segment 259AE000:
Amounts of identified storage: Free:00000E18 Allocated:000001C8 Total:00000FE0
Number of identified areas: Free: 1 Allocated: 1 Total: 2
00000000 bytes of storage were not accounted for.
No errors were found while processing this heap segment.
This is the last heap segment in the current heap.

Heap report sections of the LEDATA output
The Heap Report sections of the LEDATA output provide information for each heap segment in the dump. The detailed heap segment reports include information on the free storage tree in the heap segments, the allocated storage elements, and the cause of heap management data structure problems.

Table 22. Contents of the Heap report sections of LEDATA output

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [1] Free Storage Tree Report | Within each heap segment, Language Environment keeps track of unallocated storage areas by chaining them together into a tree. Each free area represents a node in the tree. Each node contains a header, which points to its left and right child nodes. The header also contains the length of each child. The LEDA HEAP option formats the free storage tree within each heap, and validates all node addresses and lengths within each node. Each node address is validated to ensure that it:  
  • Falls on a doubleword boundary  
  • Falls within the current heap segment  
  • Does not point to itself  
  • Does not point to a node that was previously traversed  
  
  Each node length is validated to ensure that it:  
  • Is a multiple of 8  
  • Is not larger than the heap segment length  
  • Does not cause the end of the node to fall outside of the current heap segment  
  • Does not cause the node to overlap another node  
  
  If the formatter finds a problem, then it will place an error message describing the problem directly after the formatted line of the node that failed validation. |
| [2] Heap Segment Map Report | The LEDA HEAP option produces a report that lists all of the storage areas within each heap segment, and identifies the area as either allocated or freed. For each allocated area the contents of the first X'20 ' bytes of the area are displayed in order to help identify the reason for the storage allocation.  
  
  Each allocated storage element has an 8 byte prefix used by Language Environment to manage the area. The first fullword contains a pointer to the start of the heap segment. The second fullword contains the length of the allocated storage element. The formatter validates this header to ensure that its heap segment pointer is valid. The length is also validated to ensure that it:  
  • Is a multiple of 8  
  • Is not zero  
  • Is not larger than the heap segment length  
  • Does not cause the end of the element to fall outside of the current heap segment  
  • Does not cause the element to overlap a free storage node  
  
  If the heap_free_value of the STORAGE runtime option was specified, then the formatter also checks that the free storage within each free storage element is set to the requested heap_free_value. If a problem is found, then an error message describing the problem is placed after the formatted line of the storage element that failed validation. |
Diagnosing heap damage problems
Heap storage errors can occur when an application allocates a heap storage element that is too small for it to use, and therefore, accidently overlays heap storage. If this situation occurs then some of the typical error messages generated are:
• The node address does not represent a valid node within the heap segment
• The length of the segment is not valid, or
• The heap segment pointer is not valid.

If one of the above error messages is generated by one of the reports, then examine the storage element that immediately precedes the damaged node to determine if this storage element is owned by the application program. Check the size of the storage element and ensure that it is sufficient for the program's use. If the size of the storage element is not sufficient then adjust the allocation size.

If an error occurs indicating that the node's pointers form a circular loop within the free storage tree, then check the Free Storage Tree Report to see if such a loop exists. If a loop exists, then contact the IBM support center for assistance because this may be a problem in the Language Environment heap management routines.

Additional diagnostic information regarding heap damage can be obtained by using the HEAPCHK runtime option. This option provides a more accurate time perspective on when the heap damage actually occurred, which could help to determine the program that caused the damage. For more information on HEAPCHK, see z/OS Language Environment Programming Reference.

Diagnosing storage leak problems
A storage leak occurs when a program does not return storage back to the heap after it has finished using it. To determine if this problem exists, do one of the following:
• The call-level suboption of the HEAPCHK runtime option causes a report to be produced in the CEEDUMP. Any still-allocated (that is, not freed) storage identified by HEAPCHK is listed in the report, along with the corresponding traceback. This shows any storage that wasn’t freed, as well as all the calls that were involved in allocating the storage. For more information about the HEAPCHK runtime option, see z/OS Language Environment Programming Reference.
• Examine the Heap Segment Map report to see if any data areas, within the allocated storage elements, appear more frequently than expected. If they do, then check to see if these data areas are still being used by the application program. If the data areas are not being used, then change the program to free the storage element after it is done with it.

Diagnosing heap fragmentation problems
Heap fragmentation occurs when allocated storage is interlaced with many free storage areas that are too small for the application to use. Heap fragmentation could indicate that the application is not making efficient use of its heap storage. Check the Heap Segment Map report for frequent free storage elements that are interspersed with the allocated storage elements.

Understanding the heap pools LEDATA output
The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap pools report when HEAPPPOOLS is ON. The detailed heap pools report is useful when trying to find potential damaged cells because it provides very specific
Heap Pool Report

Data for pool 1:

POOLDATA: 25C1E000
+000000 POOL_INDEX:00000001 INPUT_CELL_SIZE:00000008
+000008 CELL_SIZE:00000028 INPUT_PERCENT:00000001
+000010 CELL_POOL_SIZE:00000140 CELL_POOL_NUM:00000001
+000018 POOL_LATCH_ADDR:25C45908 POOL_INDEX:00000001
+000020 LAST_CELL:25C45340 NEXT_CELL:25C45328
+000028 Q_CONTROL_INFO:000000319 Q_FIRST_CELL:25C45600
+000030 POOL_NUM_GET_TOTAL:00000190 POOL_NUM_FREE:00000190
+000038 POOL_EXTENTS_ANCHOR:25C45688 POOL_INDEX_SAME_SIZE:01
+00003D POOL_INDEX_SIZE:00000000 POOL_NUM_SAME_SIZE:01
+000040 POOL_TRACE_TABLE:25C45600


EXTENT: 25C45688

To display entire pool extent: IP LIST 25C45808 LEN(X'00200008') ASID(X'0200')

[1] Verifying free chain for pool: 1...

No errors were found while processing free chain.

Summary of analysis for Pool 1:
Number of cells: 123 Free: 19 Allocated: 0 Total Used: 20

No errors were found while processing this Pool.

Data for pool 2:

POOLDATA: 25C1F000
+000000 POOL_INDEX:00000002 INPUT_CELL_SIZE:00000002
+000008 CELL_SIZE:00000028 INPUT_PERCENT:00000001
+000010 CELL_POOL_SIZE:00000140 CELL_POOL_NUM:00000001
+000018 POOL_LATCH_ADDR:25C45688 POOL_INDEX:00000001
+000020 LAST_CELL:25C45340 NEXT_CELL:25C45328
+000028 Q_CONTROL_INFO:000000319 Q_FIRST_CELL:25C45600
+000030 POOL_NUM_GET_TOTAL:00000190 POOL_NUM_FREE:00000190
+000038 POOL_EXTENTS_ANCHOR:25C45688 POOL_INDEX_SAME_SIZE:01
+00003D POOL_INDEX_SIZE:00000000 POOL_NUM_SAME_SIZE:01
+000040 POOL_TRACE_TABLE:25C45600


EXTENT: 25C45688

To display entire pool extent: IP LIST 25C45808 LEN(X'00200008') ASID(X'0200')

[1] Verifying free chain for pool: 2...

No errors were found while processing free chain.

Summary of analysis for Pool 2:
Number of cells: 123 Free: 19 Allocated: 0 Total Used: 20

No errors were found while processing this Pool.

Data for pool 3:

POOLDATA: 25C1F000
+000000 POOL_INDEX:00000003 INPUT_CELL_SIZE:00000008
+000008 CELL_SIZE:00000028 INPUT_PERCENT:00000001
+000010 CELL_POOL_SIZE:00000140 CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:25C45688 POOL_INDEX:00000002
+000020 LAST_CELL:25C45340 NEXT_CELL:25C45328
+000028 Q_CONTROL_INFO:000000319 Q_FIRST_CELL:25C45600
+000030 POOL_NUM_GET_TOTAL:00000190 POOL_NUM_FREE:00000190
+000038 POOL_EXTENTS_ANCHOR:25C45688 POOL_INDEX_SAME_SIZE:01
+00003D POOL_INDEX_SIZE:00000000 POOL_NUM_SAME_SIZE:01
+000040 POOL_TRACE_TABLE:25C45600


EXTENT: 25C45688

To display entire pool extent: IP LIST 25C45808 LEN(X'00200002') ASID(X'0200')

[1] Verifying free chain for pool: 3...

No errors were found while processing free chain.

Summary of analysis for Pool 3:
Number of cells: 123 Free: 19 Allocated: 0 Total Used: 20

No errors were found while processing this Pool.
### Summary of analysis for Pool 3:

Number of cells: Unused: 1 Free: 3 Allocated: 4 Total Used: 8

No errors were found while processing free chain.

### Data for pool 4:

**POOLDATA: 25C1F100**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+00000</td>
<td>POOL_INDEX</td>
<td>00000004</td>
</tr>
<tr>
<td>+00008</td>
<td>CELL_SIZE</td>
<td>00000108</td>
</tr>
<tr>
<td>+00010</td>
<td>CELL_POOL_SIZE</td>
<td>00000420</td>
</tr>
<tr>
<td>+00018</td>
<td>POOL_LATCH_ADDR</td>
<td>25C54C10</td>
</tr>
<tr>
<td>+00020</td>
<td>LAST_CELL</td>
<td>25C462E8</td>
</tr>
<tr>
<td>+00028</td>
<td>Q_CONTROL_INFO</td>
<td>0000063E</td>
</tr>
<tr>
<td>+00030</td>
<td>POOL_NUM_GET_TOTAL</td>
<td>00000320</td>
</tr>
<tr>
<td>+00038</td>
<td>POOL_INDEX_SAME_SIZE</td>
<td>04</td>
</tr>
<tr>
<td>+00030</td>
<td>POOL_INDEX_SIZE</td>
<td>04</td>
</tr>
<tr>
<td>+00040</td>
<td>POOL_TRACE_TABLE</td>
<td>25C6600C</td>
</tr>
</tbody>
</table>

### Summary of analysis for Pool 4:

Number of cells: Unused: 2 Free: 2 Allocated: 0 Total Used: 4

No errors were found while processing free chain.

### Data for pool 5.1:

**POOLDATA: 25C1F200**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+00000</td>
<td>POOL_INDEX</td>
<td>00000005</td>
</tr>
<tr>
<td>+00008</td>
<td>CELL_SIZE</td>
<td>00000400</td>
</tr>
<tr>
<td>+00010</td>
<td>CELL_POOL_SIZE</td>
<td>00000400</td>
</tr>
<tr>
<td>+00018</td>
<td>POOL_LATCH_ADDR</td>
<td>25C54C24</td>
</tr>
<tr>
<td>+00020</td>
<td>LAST_CELL</td>
<td>25E4B4CA8</td>
</tr>
<tr>
<td>+00028</td>
<td>Q_CONTROL_INFO</td>
<td>0000010D</td>
</tr>
<tr>
<td>+00030</td>
<td>POOL_NUM_GET_TOTAL</td>
<td>00000002F2</td>
</tr>
<tr>
<td>+00038</td>
<td>POOL_INDEX_SAME_SIZE</td>
<td>05</td>
</tr>
<tr>
<td>+00030</td>
<td>POOL_INDEX_SIZE</td>
<td>05</td>
</tr>
<tr>
<td>+00040</td>
<td>POOL_TRACE_TABLE</td>
<td>25D160E0</td>
</tr>
</tbody>
</table>

### Summary of analysis for Pool 5.1:

Number of cells: Unused: 3 Free: 3 Allocated: 2 Total Used: 8

No errors were found while processing free chain.
Summary of analysis for Pool 5.2:
Number of cells: Unused: 1 Free: 3 Allocated: 0 Total Used: 4
No errors were found while processing free chain.

Data for pool 5.3:
POOLDATA: 25C47C40
+000000 POOL_INDEX:00000006 INPUT_CELL_SIZE:00000400
+000008 CELL_SIZE:00000408 INPUT_PERCENT:00000099
+000010 CELL_POOL_SIZE:000001020 CELL_POOL_NUM:000000004
+000018 POOL_LATCH_ADDR:25C54C24 POOL_EXTENTS:00000001
+000020 LAST_CELL:25C47518 NEXT_CELL:25C47018
+000028 Q_CONTROL_INFO:00000100 Q_FIRST_CELL:25C47640
+000030 POOL_NUM_GET_TOTAL:00000000 POOL_NUM_FREE:00000003
+000038 POOL_EXTENTS_ANCHOR:25C463F8 POOL_INDEX_SIZE:05
+000030 POOL_INDEX_SIZE:05 POOL_NUM_SAME_SIZE:05
+000040 POOL_TRACE_TABLE:25DA6100
EXTENT: 25C463F8
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire extent: IP LIST 25C463F8 LEN(X'00001028') ASID(X'0020')
25C46400: Free storage cell. To display: IP LIST 25C46400 LEN(X'00000040') ASID(X'0020')
25C46808: Free storage cell. To display: IP LIST 25C46808 LEN(X'00000040') ASID(X'0020')
25C46C10: Free storage cell. To display: IP LIST 25C46C10 LEN(X'00000040') ASID(X'0020')

[1] Verifying free chain for pool: 5.2...
No errors were found while processing free chain.
Summary of analysis for Pool 5.2:
Number of cells: Unused: 1 Free: 3 Allocated: 0 Total Used: 4
No errors were found while processing this Pool.

Data for pool 5.4:
POOLDATA: 25C47F50
+000000 POOL_INDEX:00000007 INPUT_CELL_SIZE:00000400
+000008 CELL_SIZE:00000408 INPUT_PERCENT:00000001
+000010 CELL_POOL_SIZE:000001020 CELL_POOL_NUM:000000004
+000018 POOL_LATCH_ADDR:25C54C24 POOL_EXTENTS:00000001
+000020 LAST_CELL:25C48048 NEXT_CELL:25C48048
+000028 Q_CONTROL_INFO:00000100 Q_FIRST_CELL:25C48040
+000030 POOL_NUM_GET_TOTAL:00000000 POOL_NUM_FREE:00000003
+000038 POOL_EXTENTS_ANCHOR:25C47428 POOL_INDEX_SIZE:03
+000030 POOL_INDEX_SIZE:03 POOL_NUM_SAME_SIZE:03
+000040 POOL_TRACE_TABLE:25DA6104
EXTENT: 25C47428
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire extent: IP LIST 25C47428 LEN(X'00001028') ASID(X'0020')
25C47430: Free storage cell. To display: IP LIST 25C47430 LEN(X'00000040') ASID(X'0020')
25C47838: Free storage cell. To display: IP LIST 25C47838 LEN(X'00000040') ASID(X'0020')
25C47C40: Free storage cell. To display: IP LIST 25C47C40 LEN(X'00000040') ASID(X'0020')

[1] Verifying free chain for pool: 5.4...
No errors were found while processing free chain.
Summary of analysis for Pool 5.4:
Number of cells: Unused: 1 Free: 3 Allocated: 0 Total Used: 4
No errors were found while processing this Pool.

Data for pool 5.5:
POOLDATA: 25C4F600
+000000 POOL_INDEX:00000009 INPUT_CELL_SIZE:00000400
+000008 CELL_SIZE:00000408 INPUT_PERCENT:00000001
+000010 CELL_POOL_SIZE:000001020 CELL_POOL_NUM:000000004
+000018 POOL_LATCH_ADDR:25C54C24 POOL_EXTENTS:00000001
+000020 LAST_CELL:25C49060 NEXT_CELL:25C49060
+000028 Q_CONTROL_INFO:00000100 Q_FIRST_CELL:25C49060
+000030 POOL_NUM_GET_TOTAL:00000000 POOL_NUM_FREE:00000003
+000038 POOL_EXTENTS_ANCHOR:25C49058 POOL_INDEX_SIZE:03
+000030 POOL_INDEX_SIZE:03 POOL_NUM_SAME_SIZE:03
+000040 POOL_TRACE_TABLE:25D76120
EXTENT: 25C49058
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire extent: IP LIST 25C49058 LEN(X'00001028') ASID(X'0020')
25C49060: Free storage cell. To display: IP LIST 25C49060 LEN(X'00000040') ASID(X'0020')
25C49468: Free storage cell. To display: IP LIST 25C49468 LEN(X'00000040') ASID(X'0020')
25C49870: Free storage cell. To display: IP LIST 25C49870 LEN(X'00000040') ASID(X'0020')

[1] Verifying free chain for pool: 5.5...
No errors were found while processing free chain.
Summary of analysis for Pool 5.5:
Number of cells: Unused: 1 Free: 3 Allocated: 0 Total Used: 4
No errors were found while processing this Pool.
Heap pools report sections of the LEDATA output

The heap pools report provides information about the following items:

- Each cell pool.
- The free chain associated with every qpcb pool data area, and all the free and allocated cells in the extent chain.
- Errors found when the cells are validated.

Table 23. Contents of heap pools report sections of LEDATA output

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Free Chain Validation</td>
<td>Within each cell pool, Language Environment keeps track of unallocated cells by chaining them together. The LEDATA HEAP option validates the free chain within each cell pool. It verifies that the cell pointer is within a valid extent and that the cell pool number is valid. If the formatter finds a problem, it will place an error message describing the problem directly after the formatted line of the cell that failed validation.</td>
</tr>
</tbody>
</table>
Table 23. Contents of heap pools report sections of LEDATA output (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2] Heap Pool Extent Mapping Report</td>
<td>The LEDATA HEAP option produces a report that lists all of the cells within each pool extent, and identifies the cells as either allocated or freed. For each allocated cell, the contents of the first X'20' bytes of the area are displayed to identify the reason for the storage allocation. The formatter validates if cell pool number in header is correct.</td>
</tr>
</tbody>
</table>

Understanding the heap pools trace LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap pools trace report when the HPT option is used. The argument value is the ID of the pool to be formatted in the report. Table 24 on page 126 describes the contents of the report.

HPT(3)
***********************************************************
LANGUAGE ENVIRONMENT DATA
***********************************************************

Language Environment Product 04 V01 R10.00

[1] HEAPPOOLS Trace Table


Type: FREE  Cell Address: 25E91AC0  Cpid: 01  Tcb: 008AFCF0

[4] CALL NAME  CALL ADDRESS  CALL OFFSET
GetStorage::~GetStorage() 25E53360 00000088
foo8() 25E53598 00000086
foo7() 25E53678 0000005A
foo6() 25E536F0 0000005A
foo5() 25E53768 0000005A
foo4() 25E537E0 0000005A
foo3() 25E53858 0000005A
foo2() 25E538D0 0000005A
foo1() 25E53948 0000005A
thread 25E53A50 00000000

Timestamp: 2008/03/14 14:10:22.614087
Type: FREE  Cell Address: 25E51BD0  Cpid: 01  Tcb: 008AFCF0

CALL NAME  CALL ADDRESS  CALL OFFSET
GetStorage::~GetStorage() 25E53360 00000088
foo9() 25E53430 00000086
foo8() 25E53598 0000009A
foo7() 25E53678 0000005A
foo6() 25E536F0 0000005A
foo5() 25E53768 0000005A
foo4() 25E537E0 0000005A
foo3() 25E53858 0000005A
foo2() 25E538D0 0000005A
foo1() 25E53948 00000000

Timestamp: 2008/03/14 14:10:22.614034
Type: FREE  Cell Address: 25E51BD0  Cpid: 01  Tcb: 008AF6A0

CALL NAME  CALL ADDRESS  CALL OFFSET
GetStorage::~GetStorage() 25E53360 00000088
foo8() 25E53598 00000086
foo7() 25E53678 0000005A
foo6() 25E536F0 0000005A
foo5() 25E53768 0000005A
foo4() 25E537E0 0000005A
foo3() 25E53858 0000005A
foo2() 25E538D0 0000005A
foo1() 25E53948 00000000

124  z/OS Language Environment Debugging Guide
foo1() 25E53948 00000000
thread 25E53A50 00000000

Timestamp: 2008/03/14 14:10:22.614032
Type: FREE Cell Address: 25E91C58 Cpuid: 01 Tcb: 008AFA60
CALL NAME CALL ADDRESS CALL OFFSET
GetStorage::GetStorage() 25E53360 0000000B
foo9() 25E53430 00000086
foo8() 25E5359B 0000009A
foo7() 25E5367B 0000005A
foo6() 25E536F0 0000005A
foo5() 25E5376B 0000005A
foo4() 25E537E0 0000005A
foo3() 25E53858 0000005A
foo2() 25E538D0 0000005A
foo1() 25E53948 00000000

Timestamp: 2008/03/14 14:10:22.614030
Type: GET Cell Address: 25E91C58 Cpuid: 01 Tcb: 008AFA60
CALL NAME CALL ADDRESS CALL OFFSET
GetStorage::GetStorage() 25E53298 0000000C
foo9() 25E5339B 0000008C
foo8() 25E53430 00000086
foo7() 25E5359B 0000009A
foo6() 25E5367B 0000005A
foo5() 25E536F0 0000005A
foo4() 25E5376B 0000005A
foo3() 25E53858 0000005A
foo2() 25E538D0 0000005A
foo1() 25E53948 00000000

Chapter 3. Using Language Environment debugging facilities 125
foo3()  25E53858   0000005A
foo2()  25E538D0   0000005A
foo1()  25E53948   0000005A
thread  25E53A50   00000000

Timestamp: 2008/03/14 14:10:22.593976
Type: GET  Cell Address: 25E91A38  Cpuid: 01  Tcb: 008AFE88
CALL NAME  CALL ADDRESS  CALL OFFSET
CEEOPMI  0601F218  00000822
CEEOPC  0600C2C8  00000CEE
pthread_create  0650FE40  00000632
main  25E53800  000000EE
EDCZMINV  064C2106  00000000

Timestamp: 2008/03/14 14:10:22.557633
Type: GET  Cell Address: 25E919B0  Cpuid: 01  Tcb: 008AFE88
CALL NAME  CALL ADDRESS  CALL OFFSET
CEEOPMI  0601F218  00000822
pthread_mutex_init  06428B90  00000094
pthread_create  0650FE40  000002FC
main  25E53800  000000EE
EDCZMINV  064C2106  00000000

Timestamp: 2008/03/14 14:10:22.551547
Type: GET  Cell Address: 25E91928  Cpuid: 01  Tcb: 008AFE88
CALL NAME  CALL ADDRESS  CALL OFFSET
CEEOPMI  0601F218  00000822
pthread_mutex_init  06428B90  00000094
pthread_create  0650FE40  0000026C
main  25E53800  000000EE
EDCZMINV  064C2106  00000000

Timestamp: 2008/03/14 14:10:22.544328
Type: GET  Cell Address: 25E918A0  Cpuid: 01  Tcb: 008AFE88
dllinit  0622FBF8  0000009E
CEEZIDT  060C4B08  00000000

Table 24. Contents of heap pools trace section of LEDATA output

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2] Pool Information</td>
<td>Information includes the number of the pool (POOLID) that is currently being formatted, the ASID, and the number of entries formatted and the total number of entries taken. <strong>Note:</strong> The trace wraps for each poolid after a specific number of entries. The number of entries is controlled by the HEAPCHK runtime option.</td>
</tr>
<tr>
<td>[3] Timestamp</td>
<td>The time this trace entry was taken. The trace entries are formatted in reverse order (most recent trace entry first).</td>
</tr>
</tbody>
</table>
| [4] Trace Table Entry contents | The individual trace entry:  
  * The TYPE - GET or FREE.  
  * The Cell within the pool being acted upon.  
  * The CPU and TCB which requested or freed the cell.  
  * A traceback at the time of the request. The number of entries in this traceback is limited by the HEAPCHK runtime option. |
Understanding the C/C++-specific LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of C/C++-specific control blocks from a system dump when the COMP(C), COMP(ALL), or ALL parameter is specified and C/C++ is active in the dump. The following example illustrates the C/C++-specific output produced. Figure 8 on page 48 and Table 25 on page 145 describe the information contained in the formatted output. Ellipses are used to summarize some sections of the dump. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.

Chapter 3. Using Language Environment debugging facilities
Exiting CRTL Environment Data

********************************************************************************
CRTL I/O CONTROL BLOCKS
********************************************************************************
CIO: 20C0E098
+000000  EYE:CIO SIZE:00000090 PTR:00000000 FLG1:09
+000000  FLG2:ED FLG3:00 FLG4:00 DUMMYF:20C0E128
+000014  EDZ24:00000000 FCBSTART:2135A040 DUMMYFCB:20C0E148
+000020  MFCBSTART:2135A470 IOANYLIST:2135A000
+000028  IOBELOWLIST:00014000 FCBDDLIST:00000000
+000030  PERRORBUF:20C0DF60 TMPCOUNTER:00000000
+000038  TEMPMEM:00000000 PROMPTBUF:00000000
+000044  IOEXITS:000127C8 TERMINALCHAIN:00000000
+00004C  VANCHOR:00000000 XTI:00000000 ENOWP24:20F484A0
+000058  MAXNUMDESCRPS:00000000 DESCARRAY:00000000
+000064  TEMPFILENUM:00000000 CSS:00000000 DUMMY_NAME:........
+000074  HOSTNAME_CACHE:00000000 HOSTADDR_CACHE:00000000
+00007C  IO31:20E62508
+000080  LAST_FD_CLOSE:00000000 00000000 00000000 00000000
+000090  IGET64:213244E8 IFREE64:21323D80

FFIL: 2135A020
+000000  MARKER1:AFCB FILE:00000000 __FP:2135A040
+000000  MARKER2:AFCBAFCB FF_FLAGS:01000000
+00000C  FCBMUTEX:00000000 THREADID:00000000
+000014  HFSF:2135A174 REPOS:2135A174
+000018  GETPOS:2135A1F8 CLOSE:2135A268
+00001C  RDLL_INDEX:09 USERBUF:00000000 LRECL:00000000
+000024  BLKSIZE:00000000 BUFSIZE:00000000
+00002C  COUNTIN:00000000 COUNTOUT:00000000
+000030  BUFSIZE:00000000 NAME:........ NEXT:2135A268
+000038  PARENT:2135A040 CHID:00000000
+00003C  DNAME:........ FD:00000000 DEVTYPE:09 FCBTYPE:007C
+000044  FSCE:2135A174 UNGETBUF:2135A174 REPOS:212FD080
+000058  GETPOS:2132FC50 CLOSE:21318C78 FLUSH:212F86C0
+000064  UTILILITY:21318870 USERBUF:00000000 LRECL:00000000
+000070  BLKSIZE:00000000 BUFSIZE:00000000 BUF:00000000
+000078  UNGETCOUNT:00000000 ENDODATA:00000000 SAVEDBUF:00000000
+000084  CURSOR:00000000 ENDODATA:00000000 SAVEDBUF:00000000
+000090  REALCOUNTIN:00000000 REALCOUNTOUT:00000000
+000098  POSMAJOR:FFFFFFF SAVERMAJOR:00000000
+0000A4  POSMINOR:00000000 SAVERMINOR:00000000 STATE:0000
+0000A8  SAVSTATE:0000 EXITFTELL:00000000 EXITUNGETC:00000000
+0000B4  DBSTART:00000000 UTILITYAREA:00000000
+0000BC  INTERCEPT:00000000 FLG5:01900400 00001300
+0000CC  DBCSTART:00000000 FCB_CPCB:20C0E128
+0000D0  READGLUE:58FF0008 07FF0000 WRITE:212FD418
+0000D0  WRITEGLUE:58FF0008 07FF0000 WRITE:212FD220
+0000DC  WADDR_WSA:00000000 WWSA:00000000
+0000F0  WCEESG003:00000000 WCEESG003:00000000
+000108  RCCESG003:00000000 WWSA:00000000

FFIL: 2135A248
+000000  MARKER1:AFCB FILE:00000000 __FP:2135A040
+000000  MARKER2:AFCB AFCB FF_FLAGS:01000000
+000004  FCBMUTEX:00000000 THREADID:00000000
+000008  MAXNUMDESCRPS:00000000 DESCARRAY:00000000
+00000C  DUMMYNAME:........
+000014  EDZ24:00000000 FCBSTART:2135A040 DUMMYFCB:20C0E148
+000020  MFCBSTART:2135A470 IOANYLIST:2135A000
+000028  IOBELOWLIST:00014000 FCBDDLIST:00000000
+000030  PERRORBUF:20C0DF60 TMPCOUNTER:00000000
+000038  TEMPMEM:00000000 PROMPTBUF:00000000
+000044  IOEXITS:000127C8 TERMINALCHAIN:00000000
+00004C  VANCHOR:00000000 XTI:00000000 ENOWP24:20F484A0
+000058  MAXNUMDESCRPS:00000000 DESCARRAY:00000000
+000064  DUMMYNAME:........
+000074  HOSTNAME_CACHE:00000000 HOSTADDR_CACHE:00000000
+00007C  IO31:20E62508
+000080  LAST_FD_CLOSE:00000000 00000000 00000000 00000000
+000090  IGET64:213244E8 IFREE64:21323D80

File name: /u/charum/b235/in.txt
Chapter 3. Using Language Environment debugging facilities

File name: CHARUM.A.B

File name: CHARUM.B.C
Chapter 3. Using Language Environment debugging facilities 133
Chapter 3. Using Language Environment debugging facilities
+0000C2 LASTPOS:00000000  NEMPOS:00000000  READFUNCNUM:00000002
+000038 WRITEFUNCNUM:00000000  FCBD:2135B370  PARENT:2135B578
+000048 FLAGS1:00200000  DCBERR:00000000  DCB1::2135B5E0
+000050 OSIO_VOLSEQ:0000  OSIO_NEVOLSEQ:0000  OSIO_EXT:00000000
+000058 OSIO_HIGHVOL:0000  APPENDEDLASTVOLSEQ:0000
+00005C OSIO_JFCS:00000000

DCB: 00012880
+000000 DCBRELAD:2135B5E0  DCBFAD:0000000000  EA000B00
+000014 DCBFUNFO:00  DCBSRG1:40  DCBERRAD:00000000  DCBRECPOM:C0
+000025 DCBEXLS:0127D0  DCBDONAM:......fd  DCBMACR1:66
+000033 DCMACR2:88  DCBSYNAD:000000  DCBBK1:1800  DCBNEP:01
+000052 DCBNEP:0000

DCB: 2135B5E0
+000000 DCBED:DCBE  DCBLEN:0038  RESERVED:0000
+000008 DCBDRM:00012880  DCBENDL:00000000  DCBFLG1:C0
+000011 DCBFLG2:88  DCBENDR:0000  DCBFLG3:00
+000024 DCBBSE:00000000  DCBBEEDA:20E6269A
+00002C DCBESYNA:20E6263C  MULTSDN:00

JFCB: 00012888
+000000 JFCBSDM:CHURM. F.G
+00000C JFCBLENM: JFCBTS:M: 80  JFCSAM:001402A
+000006 JFCBVLEQ:0000  JFCBND1:00  JFCBINDD:40
+000008 JFCBFUNFO:00  JFCDSRG1:00  JFCDSRG2:00
+000006 JFCCF:0000  JFCLRECL:0000  JFCCF:00
+000004 JFCCBND1:00  JFCBND2:00  JFCCF:0000
+000004 JFCBEXTL:00  JFCBEXAD:0000AF  JFCMBL:00
+0000AE JFCBEXTT:01

FILE: 2135B620
+000000 MARKER1:FCB  FILE:00000000  _FP:2135B640
+00000C MARKER2:FCBACF  FF_FLAGS:01000000
+000014 FCBCMUTEX:00000000  THREADID:00000000  00000000

File name: CHURM. G.H

FBC: 2135B640
+000000 BUFTR:00000000  COUNTIN:00000000  COUTOUT:00000000
+00000C READFUNC:2135B710  WRITEFUNC:2135B730  FLAGS1:8000
+000016 DEPTH:0000  NAME:2135B7FC  _LENGTH:00000000
+000020 BUFSIZE:00000004  MEMBER:......  NEXT:2135B910
+000030 PREV:2135B370  PARENT:2135B640  CHILD:00000000
+00003C DDDNAME:SY500015  FD:FFFFF  DEVTYPE:00  FCBTYPE:0041
+00004C FSE:2135B774  UNGETBUF:2135B774  REPSE:20F5F2A0
+000050 GETPOS:20F5F230  CLOSE:000090  FLUSH:20F5F3A0
+000054 UTILITY:20F5F160  USERBUF:00000000  LRECL:00000050
+000070 BLSIZE:00000050  REALBUPTR:00000000
+000078 VNECOUNT:00000000  BUSIZE:00000051  BUF:00000000
+0000B4 CURSOR:00000000  ENDOFDATA:00000000  SAVEDBUF:00000000
+000090 RECLCOUNT:00000000  RECLCOUNT1:00000000
+000098 POSMAJOR:00000000  SAVEMAJOR:00000000
+0000A0 POSMINOR:00000000  SAVEMINOR:00000000  STATE:0000
+0000A0 SAVESTATE:0000  EXITFTELL:20F482A0  EXITUNGETC:20F481E8
+0000B4 DBSTART:00000000  UTILITYAREA:00000000
+0000C0 INTERCEPT:00000000  FLAGS2:02120020  40488100
+0000D8 DCBSTATE:0000  FCB_CPCB:20C0D6A8
+0000D0 REAGLUE:5BF0000  O7F0000 READ:20F485B0
+0000D0 RADDR_MSA:00000000  GETFNR:00000000  RDINDEX:00000000
+0000E8 RECESG003:00000000  RiMS:00000000
+0000F0 WRITEFNR:5BF0000  O7F0000 WRITE:20F4856B
+0000FC WADDR_MSA:00000000  GETFNR:00000000  WDLL_INDEX:00000000
+000198 WCESG2003:00000000  WMSA:00000000

OSNS: 2135B6774
+000000 OSNS_EYE:OSNS  READ:20F48580  WRITE:20F48B8B
+00000C REPOS:20F5F208  GETPOS:20F48330  CLOSE:20F48930
+000018 FLUSH:20F4B0C0  UTILITY:20F4B060  EXITFTELL:20F482A8
+000024 EXITUNGETC:20F481E8  OSIBLK:2135B5848
+00002C NEWLINEPTR:00000000  RECLLENGTH:00000050  FLAGS8:81000000

OS1: 2135B9984
+000000 OS1_EYE:OS1  DBNW:00012C00  DCBRU:00000000
+00000C JFCB:00012D08  CURMBUF:00000000  MBFSCOUNT:00000001
+000018 READMB:00000000  CURBLK:FFFFFFFF
+00002C LASTBLK:00000000  BLKSPERTRK:0000  OS1_ACCESS_METHOD:02
+00002F OS1_NOSEEK_TO seek:00  FIRSTPOS:00000000
+000018 READMAX:00000000 CURBLKNUM:FFFFFFFF
+000020 LASTBLKNUM:FFFFFFFF BLKSPERTRK:0000 OSIO_ACCESS_METHOD:01
+000027 OSIO_NODEEK_TO SEEK:00 FIRSTPOS:000000100
+00002C LASTPOS:000000100 NEMPOR:0000000000 READFUNCM:00000002
+000038 WRITEFUNCM:00000005 FCB:2135B910 PARENT:2135B818
+000044 FLAGS1:85020000 DCBSRG1:00 DCBEODAD:00000000 DCBMACR1:65
+000050 DCBSIZE:00000000 APPENDEDLASTVOLSEQ:0000
+00005C OSIO_JFCBX:00000000

DCB: 00012DC0
+000000 DCBRELAD:2135BB80 DCBEODAD:00000000 DCBFDAD:00000000
+000014 DCBUNNO:00 DCBEODAD:00000000 DCBMACR1:65
+000020 DCBRECFM:0080 DCBEXLSA:0127D0 DCBDDNAM:.4....a.
+000033 DCBMACR2:B8 DCBNCP:01 DCBLRECL:0050
+000052 DCBESIZE:00000000 DCBESYNA:20E6263C MULTSDN:00

JFCB: 00012E28
+000000 JFCBDSNM:CHARUM.H.I JFCBRELAD:2135BB80 JFCBELNM:00
+000014 JFCDSCB:002214 JFCBDSCB:00 JFCLRECL:00000000
+000020 JFCBNVOL:01 JFCBNVOLS:SL8B14 JFCLRECL:00000000
+000032 JFCHVOLSEQ:00000000 JFCHVOLSEQ:00000000
+000044 JFCBEXTL:00 JFCHVOLSEQ:00000000

File name: CHARUM.I.J

FCB: 2135B88E
+000000 BUFPTR:00000000 COUNTIN:00000000 COUNTOUT:00000000
+00000C READFUNCM:2135B8CD WRITEFUNCM:2135B8CD FLASG:8000
+000020 BUFFER:00000000 NAME:2135B8DC LENGTH:00000000
+000038 BUFSIZE:00000044 MEMBER:.00. NEXT:2135B8EB
+000050 PREV:2135B910 PARENT:2135B8ED CHILD:00000000
+000068 DDNAME:SYS00017 FD:00000000 DEVTYPE:00 FCBTYPE:0036
+00007C FSCE:2135BD14 UNGETBUF:2135BD14 REPOS:20F5BFD0
+000090 GETPOS:20FA84E0 CLOSE:20FA16C0 FLUSH:20FA3FD0
+0000A8 UTILITY:20FA0550 USERBUF:00000000 LRECL:00000040
+0000B0 BLKSIZE:00000048 REALBUFPTR:00000000
+0000BC UNGETCOUNT:00000000 BUFSIZE:00000049 BUFSIZE:00000000
+0000C4 CURSOR:00000000 ENDOFDATA:00000000 SAVEDBUF:00000000
+0000D0 REALCOUNTIN:00000000 REALCOUNTOUT:00000000
+0000E0 POSMAJOR:00000000 SAVEMAJOR:00000000
+0000F0 POSMINOR:00000000 SAVEMINOR:00000000 STATE:00
+000100 SAVESTATE:0000 EXITTELL:00000000 EXITUNGETC:20F4A16B
+000118 DBSTART:00000000 UTILITYAREA:00000000
+000120 INTERCEPT:00000000 FLAGS2:02120020 20440100
+000138 DBSCSTATE:0000 FCB_CPCB:20C0DEA8
+000140 REGLUE:58FF0008 DCBRELAD:2135BB80 REGLUE:58FF0008
+000150 RADR_MSA:00000000 GETFN:00000000 RDLL_INDEX:00000000
+000168 RECSG003:00000000 RWSA:00000000
+00017C WRITEGLUE:58FF0008 WADDR_WSA:00000000
+000190 WCEESG003:00000000 WCEESG003:00000000
+0001A8 WCEESG003:00000000

OSVF: 2135B9D4
+000000 OSVF_EVE:00000000 OSVF_READ:20F4A85B0 WRITE:20F4AFD80
+00000C REPOS:20FA94AD GETPOS:20FA94AD CLOSE:20FAA16C0
+000018 FLUSH:20FAA3FD0 UTILITY:20FAA550 EXITTELL:00000000
+000024 EXITUNGETC:20F4A16B OSVJBLK:2135B8EB
+00002C SAVECURSOR:00000000 NEMLINPRT:JFFFFF
+000034 CURBLKSIZE:00000000 LASTBLKSIZE:00000000
+00003C HIGHMAJOR:00000000 MAXFTELLBLK:001FFFFF
+000044 RECIBITS:00000008 FLAGS:73100000 RECLEN:00000000
+000050 CURBLKNUM:00000000 LASTBLKNUM:00000000
+000058 OSWRITEETOC:00000000 RECCOUNTOUT:00000000

138 z/OS Language Environment Debugging Guide
Chapter 3. Using Language Environment debugging facilities 139
Chapter 3. Using Language Environment debugging facilities
Chapter 3. Using Language Environment debugging facilities 143
Dummy FCB encountered at location 20C0E148

AMRC: 20C0E770
+000000 CODE:00002170B RBA:00000000 LASTOP:000000000032
+000000 FILL_LEN:00000000 MSG_LEN:00000000
+000014 STRI:.........................................................
+000050 STRI_CONT:...................................................
+00008C PARAMO:00000000 PARAMR1:00000000
+00009C STR:................................................................
+0000DC RPLFDBMD:00000000 XBA:0000000000000000
+0000E8 AMRC_NOSEEK_TO_SEEK:00

AMRC2: 20C0E878
+000000 __ERROR2:00000000 __FILEPTR:00000000

File name: CHARUM.A.B
Exiting CRTL I/O Control Blocks
Exiting Language Environment Data

Table 25. Contents of C/C++-specific sections of LEDATA output

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] CGEN</td>
<td>Formats the C/C++-specific portion of the Language Environment common anchor area (CAA).</td>
</tr>
<tr>
<td>[2] CGENE</td>
<td>Formats the extension to the C/C++-specific portion of the Language Environment common anchor area (CAA).</td>
</tr>
</tbody>
</table>
Table 25. Contents of C/C++-specific sections of LEDATA output (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7] File Control Blocks</td>
<td>Formats the C/C++ file control block (FCB). The FCB and its related control blocks represent the information needed by each open stream. The following related control blocks are included.</td>
</tr>
<tr>
<td></td>
<td>FFIL</td>
</tr>
<tr>
<td></td>
<td>FSCE</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OSIO</td>
</tr>
<tr>
<td></td>
<td>OSIOE</td>
</tr>
<tr>
<td></td>
<td>DCB</td>
</tr>
<tr>
<td></td>
<td>DCBE</td>
</tr>
<tr>
<td></td>
<td>JFCB</td>
</tr>
<tr>
<td></td>
<td>JFCBX</td>
</tr>
<tr>
<td></td>
<td>MBUF</td>
</tr>
<tr>
<td>[8] Memory File Control Blocks</td>
<td>This section formats the C/C++ memory file control block (MFCB).</td>
</tr>
</tbody>
</table>

Understanding the COBOL-specific LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of COBOL-specific control blocks from a system dump when the COMP(COBOL), COMP(ALL) or ALL parameter is specified and COBOL is active in the dump. The following example illustrates the COBOL-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option. Table 26 on page 148 describes the information contained in the formatted output. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.

```
RUNCOM: 00049038
       +000000 IDENT:C3RUNCOM LENGTH:000002D8 FLAGS:00860000
       +000010 RU_ID:00001789 INVK_RSA:00005FB0
       +000024 MAIN_PGM_ADDR:00007658 MAIN_PGM_CLLE:00049328
```
Chapter 3. Using Language Environment debugging facilities

147
Table 2.6. Contents of COBOL-specific sections of LEDATA Output

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] RUNCOM</td>
<td>Formats the COBOL enclave-level control block (RUNCOM).</td>
</tr>
</tbody>
</table>

COBOLV5+ ENVIRONMENT DATA

CLOBEDB: 0CFE7048

RUNCOM: 000010 IDENT:CLOBEDB LENGTH:000001C INPL_MAIN:0C600000

+000014 FLAGS:80000000 MAIN_CLLE:0CFE7C0B Dummy_CLLE:0CFE9A0B

+000020 FREE_CLLE:00000000 CLLE_HASH:0CFE9608

+000028 FIRST_CLLE:074E360 EXT_DATA_LIST:00000000

+000030 EXT_FILE_BLOCK:00000000 EXT_FILESYNC@:00000000

+000038 VAR EVVT:000C3E0A 000380 ENVT NT:000E5590

+000040 CLLE:00000000 EXT_DATA:00000000

+000060 QSAM_EOD_DSP.Func:00000000 QSAM_GET_DSP.Func:00000000

+000068 QSAM_PUT_DSP.Func:00000000 QSAM_PUTX_DSP.Func:00000000

+000070 QSAM_UPUT_DSP.Func:00000000 QSAM_LBI GMT_DSP.Func:00000000

+000080 QSAM_LBI GMT_DSP.Func:00000000 QSAM_QV_DSP.Func:00000000

+000088 QSAM_VL_DSP.Func:00000000 QSAM_EOD_DSP.EXIT:00000000

+000090 QSAM_VL_DSP.EXIT:00000000 VSAM_VL_DSP.Func:00000000

+000098 VSAM_READ_DSP.Func:00000000 VSAM_WRITE_DSP.Func:00000000

+0000A0 LSEQ_READ_DSP.Func:00000000 LSEQ_WRITE_DSP.Func:00000000

+0000A8 DUMMY_CLLE:0CFE9A0B LE_UPSI:00000000

+0000C0 LE_UPSI:00000000 MTX_PGM:00000000 MTX_SYSIN:00000000

+0000C8 MTX_SYSOUT:00000000 MTX_SYSUN:00000000

+0000D0 MTX_CONSOLE:00000000 MTX_RANDOM:00000000 MTX_CLL:00000000

+0000E0 MTX_MISC_EVENTS:00000000 A THREAD STATUS:0CFE7F04

+0000F0 A THREAD_STATUS:00000000 A MTX_SYSIN:0CFE79E8

+000100 A MTX_SYSOUT:0CFE79EC A MTX_SYSUN:0CFE79F0

+000110 A MTX_CONSOLE:0CFE79F0 A MTX_RANDOM:00000000

+000120 A RANDOM SEED:0CFE7AC RANDOM SEED:00000000

+000130 LE_TRACE_FLAGS:00000000 FIRST_CAA_ENCLAVE:0C61C8E0

+000140 COBOLV5+ ENVIRONMENT DATA

+000148 V4_SYNC_FLAGS:00000000

+000150 HAS C:00000000 HAS_16XBP:00000000

+000158 DBG_API:00000000 CBO_THREAD IPT:0CFE7198 CAA_SORT:0C61C8E0

+000168 IDENT 2::E_CLOBEDB

COBTHRED: 0CFE7198

+000000 IDENT:COBTHRED FLAGS:00000000 IOSAVEAREA:0CFE7360

+000010 IONATIONALWORKAREA:0CFE7400 WSA2452:00000000 THCSIGPS:0CFE7418

+00001C XDCALP:0CFE7478 PRINTF_INV:00013A8A FAL_CACHE:0CFE84A0

+000028 INIT_CACHE:00000000 RDSQGLST:00000000
Chapter 3. Using Language Environment debugging facilities

149
The names of the control blocks have changed in Enterprise COBOL V5.1. Table 27 shows the correspondence with COBOL V4R2 and prior releases.

Table 27. Contents of COBOL-specific sections of LEDATA Output (Enterprise COBOL V5.1 and later releases)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] COBEDB</td>
<td>Corresponds to RUNCOM, formats the COBOL enclave-level control block.</td>
</tr>
<tr>
<td>[2] COBPCB</td>
<td>Corresponds to THDCOM, formats the COBOL process-level control block.</td>
</tr>
<tr>
<td>[3] COBRCCB</td>
<td>Corresponds to COBCOM, formats the COBOL region-level control block.</td>
</tr>
<tr>
<td>[4] CLLE</td>
<td>Formats the COBOL loaded program control block (same name).</td>
</tr>
<tr>
<td>[5] COBDSACB</td>
<td>Corresponds to TGT, program-level control block.</td>
</tr>
</tbody>
</table>

Understanding the PL/I-specific LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of PL/I-specific control blocks from a system dump when the COMP(PLI), COMP(ALL) or ALL parameter is specified and PL/I is active in the dump. The following example illustrates the PL/I-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option. Table 28 on page 155 describes the information contained in the formatted output. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.

************************************************************
PL/I FOR MVS & VM ENVIRONMENT DATA
************************************************************

[1] RXRCB: 00021000
+000000 1D:ZRCB LDBWC:00011338 RSP:00000000
+000010 PSM:00000000 PSRL:00000000
+000000 1D:ZPRB RCB:00021000 SYSF_FCB:00006304
+000010 MSG_FCB:00000000 PRV_INIT:0001683B
+000010 ENT_FCB:00000000 MSGF:00000000
+000030 DCLLIST:00000000 DCL_LIST:00000000
+000038 DCL_LIST_LEN:00000000
[3] TIA: 00056298
+000000 TISA:00000000 TAPC:00000000 TERA:00000000 TIMM:0000
+000000 TFL:00000000 TTTW:00000000 TEXF:00000000 TLFE:00000000
+000020 TDB:00056450 TDDS:00000000 TLWR:00000000
+00002C TASM:00000000 TSNM:........ TASR:00000000


Chapter 3. Using Language Environment debugging facilities
Table 28. Contents of PL/I-specific sections of LEDATA output

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>

Chapter 3. Using Language Environment debugging facilities 155
### Table 28. Contents of PL/I-specific sections of LEDATA output (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[10] TCA</td>
<td>Formats the Enterprise PL/I task communication control block (TCA).</td>
</tr>
<tr>
<td>[12] OCA</td>
<td>Formats the Enterprise PL/I ON communications control block (OCA).</td>
</tr>
</tbody>
</table>

### Formatting individual control blocks

In addition to the full LEDATA output, which contains many formatted control blocks, the IPCS Control block formatter can format individual Language Environment control blocks. The IPCS CBF command can be invoked from the "IPCS Subcommand Entry" screen, option 6 of the "IPCS PRIMARY OPTION MENU".

#### Syntax

```
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CBF — address — STRUCTure — ( — cbname — )</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
</tbody>
</table>
```

- **address**
  
  Address of the control block in the dump, which is determined by browsing the dump or running the LEDATA verb exit.

- **cbname**
  
  The name of the control block to be formatted. The control blocks that can be individually formatted are listed in [Table 29 on page 157](#). In general, the name of each control block is similar to that used by the LEDATA verb exit and is generally found in the control block’s eyecatcher field. However, all control block names are prefixed with “CEE” to uniquely define the Language Environment control block names to IPCS.

For example, the following command produces the output shown in [Figure 34 on page 157](#).

```
CBF 213F6848 struct(CEECAA)
```
For more information on using the IPCS CBF command, refer to the "CBFORMAT subcommand" section in z/OS MVS IPCS Commands. Table 29. Language Environment Control blocks that can be individually formatted

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEADHP</td>
<td>Additional Heap Control Block</td>
</tr>
<tr>
<td>CEECAA</td>
<td>Common Anchor Area</td>
</tr>
<tr>
<td>CEECIB</td>
<td>Condition Information Block</td>
</tr>
<tr>
<td>CEECIBH</td>
<td>Condition Information Block Header</td>
</tr>
<tr>
<td>CEECMXB</td>
<td>Message Services Block</td>
</tr>
<tr>
<td>CEEDSA</td>
<td>Dynamic Storage Area</td>
</tr>
<tr>
<td>CEEDLLF</td>
<td>DLL Failure Control Block</td>
</tr>
<tr>
<td>CEEDSATR</td>
<td>XPLINK Transition Area</td>
</tr>
<tr>
<td>CEEDSAX</td>
<td>Dynamic Storage Area (XPLINK style)</td>
</tr>
</tbody>
</table>
Table 29. Language Environment Control blocks that can be individually formatted (continued)

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEEDB</td>
<td>Enclave Data Block</td>
</tr>
<tr>
<td>CEEEENSM</td>
<td>Enclave Level Storage Management</td>
</tr>
<tr>
<td>CEEHANC</td>
<td>Heap Anchor Node</td>
</tr>
<tr>
<td>CEEHCOM</td>
<td>CEL Exception Manager Communications Area</td>
</tr>
<tr>
<td>CEEHPCB</td>
<td>Thread Level Heap Control Block</td>
</tr>
<tr>
<td>CEEHPSB</td>
<td>Heap Statistics Block</td>
</tr>
<tr>
<td>CEEMDTST</td>
<td>Message Destination</td>
</tr>
<tr>
<td>CEEMGF</td>
<td>Mapping of the Message Formatter (IBM1MGF)</td>
</tr>
<tr>
<td>CEEPBCB</td>
<td>Process Control Block</td>
</tr>
<tr>
<td>CEEPSCB</td>
<td>Program Management Control Block</td>
</tr>
<tr>
<td>CEERCB</td>
<td>Region Control Block</td>
</tr>
<tr>
<td>CEESKB</td>
<td>Stack Statistics Block</td>
</tr>
<tr>
<td>CEESMCB</td>
<td>Storage Management Control Block</td>
</tr>
<tr>
<td>CEESTKH</td>
<td>Stack Header Block</td>
</tr>
<tr>
<td>CEESTKHX</td>
<td>Stack Header Block (xplink style)</td>
</tr>
<tr>
<td>CEESTSB</td>
<td>Storage Report Statistics Block</td>
</tr>
<tr>
<td>CEETMXB</td>
<td>Thread Level Messages Extension Block</td>
</tr>
</tbody>
</table>

Controlling access to CEEDUMPs and DYNDUMPs

Since Language Environment dumps may provide detailed information about the internal processing and data used by an authorized application, Language Environment enforces security rules to ensure that the users of these applications have permission to obtain dumps generated for them. These dumps include the following:

- CEEDUMP information that is generated based on the TERMTHDACT runtime option settings TRACE, DUMP, UATRACE, UADUMP.
- Dynamic transaction dumps generated based on the DYNDUMP runtime option.
- Formatted dumps requested by programming interfaces, including CEE3DMP, csnap(), __cdump(), ctrace(), PLIDUMP.

Language Environment will suppress these dumps for authorized applications under the following conditions:

- A user is running a Language Environment application as a RACF-controlled program on a system where the IEAADBD.DMPAUTH resource has been defined, but the user has not been permitted access to this resource.
- A user is running an authorized key Language Environment application in a non-started task address space but the user has not been permitted access to the IEAADBD.DMPAKEY resource.
- A user is running a Language Environment application in a non-started task address space that has the JSCBPASS indicator on, including applications whose PPT entry specifies bypassing security protection.

When Language Environment has suppressed a dump, message CEE3880I will be written to the application’s programmer log. To allow the user to receive this
dump, the user may need to be permitted to the IEAABD.DMPAUTH or
IEAABD.DMPAKEY resource. For more information, refer to the documentation for

### Requesting a Language Environment trace for debugging

Language Environment provides an in-storage, wrapping trace facility that can
reconstruct the events leading to the point where a dump is taken. The trace
facility can record two types of events: entry and exit library calls and, if the
POSIX runtime option is set to ON, user mutex and condition variable activity
such as init, lock/unlock, and wait. Language Environment produces a trace table
in its dump report under the following conditions:

- The CEE3DMP callable service is invoked with the BLOCKS option and the
  TRACE runtime option is set to ON.
- The TRACE runtime option is set to NODUMP and the TERMTHDACT runtime
  option is set to DUMP, UADUMP, TRACE, or UATRACE.
- The TRACE runtime option is set to DUMP (the default).

For more information about the CEE3DMP callable service, the TERMTHDACT
runtime option, or the TRACE runtime option, see [z/OS Language Environment
Programming Reference](https://www.ibm.com).

The TRACE runtime option activates Language Environment runtime library
tracing and controls the size of the trace buffer, the type of trace events to record,
and it determines whether a dump containing only the trace table should be
unconditionally taken when the application (enclave) terminates. The trace table
contents can be written out either upon demand or at the termination of an
enclave.

The contents of the Language Environment dump depend on the values set in the
TERMTHDACT runtime option. **Table 30** summarizes the dump contents that are
generated under abnormal termination.

<table>
<thead>
<tr>
<th>TERMTHDACT value</th>
<th>Type of dump generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMTHDACT(QUIET)</td>
<td>Language Environment dump containing the trace table only</td>
</tr>
<tr>
<td>TERMTHDACT(MSG)</td>
<td>Language Environment dump containing the trace table only</td>
</tr>
<tr>
<td>TERMTHDACT(TRACE)</td>
<td>Language Environment dump containing the trace table and the traceback</td>
</tr>
<tr>
<td>TERMTHDACT(DUMP)</td>
<td>Language Environment dump containing thread/enclave/process storage and control blocks (the trace table is included as an enclave control block)</td>
</tr>
<tr>
<td>TERMTHDACT(UAONLY)</td>
<td>System dump of the user address space and a Language Environment dump that contains the trace table</td>
</tr>
<tr>
<td>TERMTHDACT(UATRACE)</td>
<td>Language Environment dump that contains traceback information, and a system dump of the user address space</td>
</tr>
<tr>
<td>TERMTHDACT(UADUMP)</td>
<td>Language Environment dump containing thread/enclave/process storage and control blocks (the trace table is included as an enclave control block), and a user address space dump</td>
</tr>
</tbody>
</table>

[Table 30. TERMTHDACT runtime option settings and dump contents produced](#)
### Table 30. TERMTHDACT runtime option settings and dump contents produced (continued)

<table>
<thead>
<tr>
<th>TERMTHDACT value</th>
<th>Type of dump generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMTHDACT(UAIMM)</td>
<td>System dump of the user address space of the original abend or program interrupt that occurred before the Language Environment condition manager processing the condition. Also contains a Language Environment dump, which contains the trace table. Under CICS, UAIMM yields UAONLY behavior. Under non-CICS, TRAP(ON,NOSPIE) must be in effect. When TRAP(ON, SPIE) is in effect, UAIMM yields UAONLY behavior. For software raised conditions or signals, UAIMM behaves the same as UAONLY.</td>
</tr>
</tbody>
</table>

Under normal termination, independent of the TERMTHDACT setting, Language Environment generates a dump containing the trace table only based on the TRACE runtime option.

Language Environment quiesces all threads that are currently running except for the thread that issued the call to CEE3DMP. When you call CEE3DMP in a multithread environment, only the current thread is dumped. Enclave- and process-related storage could have changed from the time the dump request was issued.

### Locating the trace dump

If your application calls CEE3DMP, the Language Environment dump is written to the file specified in the FNAME parameter of CEE3DMP (the default is CEEDUMP).

If your application is running under TSO or batch, and a CEEDUMP DD is not specified, Language Environment writes the CEEDUMP to the batch log (SYSOUT=* by default). You can change the SYSOUT class by specifying a CEEDUMP DD, or by setting the environment variable, `_CEE_DMPTARG=SYSOUT(x)`, where `x` is the preferred SYSOUT class.

If your application is running under z/OS UNIX and is either running in an address space you issued a `fork()` to, or if it is invoked by one of the exec family of functions, the dump is written to the hierarchical file system (HFS). Language Environment writes the CEEDUMP to one of the following directories in the specified order:

1. The directory found in environment variable `_CEE_DMPTARG`, if found
2. The current working directory, if the directory is not the root directory (`/`), the directory is writable, and the CEEDUMP path name does not exceed 1024 characters
3. The directory found in environment variable TMPDIR (an environment variable that indicates the location of a temporary directory if it is not `/tmp`)
4. The `/tmp` directory

The name of this file changes with each dump and uses the following format:

```
/path/Fname.Date.Time.Pid
```
**path**  Path determined from the above algorithm.

**Fname**  Name specified in the FNAME parameter on the call to CEE3DMP (default is CEEDUMP).

**Date**  Date the dump is taken, appearing in the format YYYYMMDD (such as 20090307 for March 7, 2009).

**Time**  Time the dump is taken, appearing in the format HHMMSS (such as 175501 for 05:55:01 p.m.).

**Pid**  Process ID the application is running in when the dump is taken.

### Using the Language Environment trace table format in a dump report

The Language Environment trace table is established unconditionally at enclave initialization time if the TRACE runtime option is set to ON. All threads in the enclave share the trace table; there is no thread-specific table, nor can the table be dynamically extended or enlarged.

### Understanding the trace table entry (TTE)

Each trace table entry is a fixed-length record consisting of a fixed-format portion (containing such items as the timestamp, thread ID, and member ID) and a member-specific portion. The member-specific portion has a fixed length, of which some (or all) can be unused. For information about how participating products use the trace table entry, see the product-specific documentation. The format of the trace table entry is as follows:

![Figure 35. Format of the trace table entry](image)

**Time**  The 64-bit value obtained from a store clock (STCK).

**Thread ID**  The 8-byte thread ID of the thread that is adding the trace table entry.

**Member ID and Flags**  Contains 2 fields:

- **Member ID**  The 1-byte member ID of the member making the trace table entry, as follows:
  - 01: CEL
  - 03: C/C++
  - 04: COBOL V5 (and later releases)
  - 05: COBOL
  - 07: Fortran
  - 08: Reserved
  - 10: PL/I
Flags  24 flags reserved for internal use.

Member Entry Type
A number that indicates the type of the member-specific trace information that follows the field. To uniquely identify the information contained in a specific TTE, you must consider Member ID as well as Member Entry Type.

Member-Specific Information
Based on the member ID and the member entry type, this field contains the specific information for the entry, up to 104 bytes. For C/C++, the entry type of 1 is a record that records an invocation of a base C runtime library function. The entry consists of the name of the invoking function and the name of the invoked function. Entry type 2 is a record that records the return from the base library function. It contains the returned value and the value of errno.

Member-specific information in the trace table entry
Global tracing is activated by using the LE=n suboption of the TRACE runtime option. This requests all Language Environment members to generate trace records in the trace table. The settings for the global trace events are:

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No global trace</td>
</tr>
<tr>
<td>1</td>
<td>Trace all runtime library (RTL) function entry and exits</td>
</tr>
<tr>
<td>2</td>
<td>Trace all RTL mutex init/destroy and lock/unlock</td>
</tr>
<tr>
<td>3</td>
<td>Trace all RTL function entry and exits, and all mutex init/destroy and lock/unlock</td>
</tr>
<tr>
<td>8</td>
<td>Trace all XPLINK/non-XPLINK transitions for AMODE 31 only. If #pragma linkage (xxxxxxxx, OS_UPSTACK) is specified, no transitions are recorded.</td>
</tr>
<tr>
<td>20</td>
<td>Trace all XPLINK/non-XPLINK transitions for AMODE 31 only.</td>
</tr>
</tbody>
</table>

When LE=1 is specified: Table 31 shows the C/C++ records that may be generated. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 209.

Table 31. LE=1 entry records

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000001</td>
<td>Base C Library function Entry</td>
</tr>
<tr>
<td>03</td>
<td>00000002</td>
<td>Base C Library function Exit</td>
</tr>
<tr>
<td>03</td>
<td>00000003</td>
<td>Posix C Library function Entry</td>
</tr>
<tr>
<td>03</td>
<td>00000004</td>
<td>Posix C Library function Exit</td>
</tr>
<tr>
<td>03</td>
<td>00000005</td>
<td>XPLINK Base or Posix C Library function Entry</td>
</tr>
<tr>
<td>03</td>
<td>00000006</td>
<td>XPLINK Base or Posix C Library function Exit</td>
</tr>
</tbody>
</table>

When LE=2 is specified: Table 32 shows the Language Environment records that may be generated.

Table 32. LE=2 entry records

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0000101</td>
<td>LT</td>
<td>A</td>
<td>Latch Acquire</td>
</tr>
</tbody>
</table>
### Table 3.2. LE=2 entry records (continued)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00000102</td>
<td>LT</td>
<td>R</td>
<td>Latch Release</td>
</tr>
<tr>
<td>01</td>
<td>00000103</td>
<td>LT</td>
<td>W</td>
<td>Latch Wait</td>
</tr>
<tr>
<td>01</td>
<td>00000104</td>
<td>LT</td>
<td>AW</td>
<td>Latch Acquire after Wait</td>
</tr>
<tr>
<td>01</td>
<td>00000106</td>
<td>LT</td>
<td>I</td>
<td>Latch Increment (Recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000107</td>
<td>LT</td>
<td>D</td>
<td>Latch Decrement (Recursive)</td>
</tr>
<tr>
<td>01</td>
<td>000002FC</td>
<td>LE</td>
<td>EUO</td>
<td>Latch unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000002FD</td>
<td>LE</td>
<td>EO</td>
<td>Latch already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>00000301</td>
<td>MX</td>
<td>A</td>
<td>Mutex acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000302</td>
<td>MX</td>
<td>R</td>
<td>Mutex release</td>
</tr>
<tr>
<td>01</td>
<td>00000303</td>
<td>MX</td>
<td>W</td>
<td>Mutex wait</td>
</tr>
<tr>
<td>01</td>
<td>00000304</td>
<td>MX</td>
<td>AW</td>
<td>Mutex acquire after wait</td>
</tr>
<tr>
<td>01</td>
<td>00000305</td>
<td>MX</td>
<td>B</td>
<td>Mutex busy (Trylock failed)</td>
</tr>
<tr>
<td>01</td>
<td>00000306</td>
<td>MX</td>
<td>I</td>
<td>Mutex increment (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000307</td>
<td>MX</td>
<td>D</td>
<td>Mutex decrement (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000315</td>
<td>MX</td>
<td>IN</td>
<td>Mutex initialize</td>
</tr>
<tr>
<td>01</td>
<td>00000316</td>
<td>MX</td>
<td>DS</td>
<td>Mutex destroy</td>
</tr>
<tr>
<td>01</td>
<td>0000031D</td>
<td>MX</td>
<td>BI</td>
<td>Shared memory lock init</td>
</tr>
<tr>
<td>01</td>
<td>0000031E</td>
<td>MX</td>
<td>BD</td>
<td>Shared memory lock destroy</td>
</tr>
<tr>
<td>01</td>
<td>0000031F</td>
<td>MX</td>
<td>BO</td>
<td>shared memory lock obtain</td>
</tr>
<tr>
<td>01</td>
<td>00000320</td>
<td>MX</td>
<td>BC</td>
<td>Shared memory lock obtain on condition</td>
</tr>
<tr>
<td>01</td>
<td>00000321</td>
<td>MX</td>
<td>BR</td>
<td>Shared memory lock release</td>
</tr>
<tr>
<td>01</td>
<td>00000324</td>
<td>MX</td>
<td>CIN</td>
<td>Call to SMC_INIT</td>
</tr>
<tr>
<td>01</td>
<td>00000325</td>
<td>MX</td>
<td>CSD</td>
<td>Call to SMC_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>00000326</td>
<td>MX</td>
<td>CSO</td>
<td>Shared resource obtain</td>
</tr>
<tr>
<td>01</td>
<td>00000327</td>
<td>MX</td>
<td>CSR</td>
<td>Shared resource release</td>
</tr>
<tr>
<td>01</td>
<td>00000328</td>
<td>MX</td>
<td>CST</td>
<td>Call to SMC_SetupToWait</td>
</tr>
<tr>
<td>01</td>
<td>00000329</td>
<td>MX</td>
<td>CSP</td>
<td>Call to SMC_POST</td>
</tr>
<tr>
<td>01</td>
<td>000004CC</td>
<td>ME</td>
<td>FFR</td>
<td>Error - Forced release (shared mutex)</td>
</tr>
<tr>
<td>01</td>
<td>000004CD</td>
<td>ME</td>
<td>FFD</td>
<td>Error - Forced decrement (shared mutex)</td>
</tr>
<tr>
<td>01</td>
<td>000004CE</td>
<td>ME</td>
<td>FBD</td>
<td>Error - BPX_SMC(DESTROY) error return</td>
</tr>
<tr>
<td>01</td>
<td>000004CF</td>
<td>ME</td>
<td>FBU</td>
<td>Error - BPX_SMC(fail) returns EBUSY</td>
</tr>
<tr>
<td>01</td>
<td>000004D0</td>
<td>ME</td>
<td>FIV</td>
<td>Error - BPX_SMC(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000004D4</td>
<td>ME</td>
<td>FDU</td>
<td>Error - Destroy failed (uninitialized) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000004D5</td>
<td>ME</td>
<td>FP</td>
<td>Error - Program check (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000004DB</td>
<td>ME</td>
<td>ESC</td>
<td>Error - BPX1SMC error return</td>
</tr>
<tr>
<td>01</td>
<td>000004DE</td>
<td>ME</td>
<td>EDL</td>
<td>Shared memory lock returns deadlock</td>
</tr>
<tr>
<td>01</td>
<td>000004DF</td>
<td>ME</td>
<td>EIV</td>
<td>Shared memory lock returns invalid</td>
</tr>
<tr>
<td>01</td>
<td>000004E0</td>
<td>ME</td>
<td>EPM</td>
<td>Shared memory lock returns eperm</td>
</tr>
<tr>
<td>Member ID</td>
<td>Record Type</td>
<td>Class</td>
<td>Event</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------</td>
<td>-------</td>
<td>-------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>01</td>
<td>000004E1</td>
<td>ME</td>
<td>EAG</td>
<td>Shared memory lock returns eagain</td>
</tr>
<tr>
<td>01</td>
<td>000004E2</td>
<td>ME</td>
<td>EBU</td>
<td>Shared memory lock returns ebusy</td>
</tr>
<tr>
<td>01</td>
<td>000004E3</td>
<td>ME</td>
<td>ENM</td>
<td>Shared memory lock returns enomem</td>
</tr>
<tr>
<td>01</td>
<td>000004E4</td>
<td>ME</td>
<td>EBR</td>
<td>Shared memory lock release error</td>
</tr>
<tr>
<td>01</td>
<td>000004E5</td>
<td>ME</td>
<td>EBC</td>
<td>Shared memory lock obtain condition error</td>
</tr>
<tr>
<td>01</td>
<td>000004E6</td>
<td>ME</td>
<td>EBO</td>
<td>Shared memory lock obtain error</td>
</tr>
<tr>
<td>01</td>
<td>000004E7</td>
<td>ME</td>
<td>EBD</td>
<td>Shared memory lock destroy error</td>
</tr>
<tr>
<td>01</td>
<td>000004E8</td>
<td>ME</td>
<td>EBI</td>
<td>Shared memory lock initialize error</td>
</tr>
<tr>
<td>01</td>
<td>000004E9</td>
<td>ME</td>
<td>EFR</td>
<td>Mutex forced release</td>
</tr>
<tr>
<td>01</td>
<td>000004EA</td>
<td>ME</td>
<td>EFD</td>
<td>Mutex forced decrement</td>
</tr>
<tr>
<td>01</td>
<td>000004EB</td>
<td>ME</td>
<td>EDD</td>
<td>Mutex destroy failed (damage)</td>
</tr>
<tr>
<td>01</td>
<td>000004EC</td>
<td>ME</td>
<td>EDB</td>
<td>Mutex destroy failed (busy)</td>
</tr>
<tr>
<td>01</td>
<td>000004ED</td>
<td>ME</td>
<td>EIA</td>
<td>Mutex initialize failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000004EE</td>
<td>ME</td>
<td>EIS</td>
<td>Mutex initialize failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000004EF</td>
<td>ME</td>
<td>EF</td>
<td>Mutex release (forced by quiesce)</td>
</tr>
<tr>
<td>01</td>
<td>000004F0</td>
<td>ME</td>
<td>EP</td>
<td>Mutex program check</td>
</tr>
<tr>
<td>01</td>
<td>000004FA</td>
<td>ME</td>
<td>EDU</td>
<td>Mutex destroy failed (uninitialized)</td>
</tr>
<tr>
<td>01</td>
<td>000004FB</td>
<td>ME</td>
<td>EUI</td>
<td>Mutex uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000004FC</td>
<td>ME</td>
<td>EUO</td>
<td>Mutex unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000004FD</td>
<td>E</td>
<td>EO</td>
<td>Mutex already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000004FE</td>
<td>ME</td>
<td>EIN</td>
<td>Mutex initialization failed (duplicate)</td>
</tr>
<tr>
<td>01</td>
<td>00000508</td>
<td>CV</td>
<td>MR</td>
<td>CV release mutex</td>
</tr>
<tr>
<td>01</td>
<td>00000509</td>
<td>CV</td>
<td>MA</td>
<td>CV reacquire mutex</td>
</tr>
<tr>
<td>01</td>
<td>0000050A</td>
<td>CV</td>
<td>MW</td>
<td>CV mutex wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050B</td>
<td>CV</td>
<td>MAW</td>
<td>CV reacquire mutex after wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050C</td>
<td>CV</td>
<td>CW</td>
<td>CV condition wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050D</td>
<td>CV</td>
<td>CTW</td>
<td>CV condition timeout</td>
</tr>
<tr>
<td>01</td>
<td>0000050E</td>
<td>CV</td>
<td>CWP</td>
<td>CV wait posted</td>
</tr>
<tr>
<td>01</td>
<td>0000050F</td>
<td>CV</td>
<td>CWI</td>
<td>CV wait interrupted</td>
</tr>
<tr>
<td>01</td>
<td>00000510</td>
<td>CV</td>
<td>CTO</td>
<td>CV wait timeout</td>
</tr>
<tr>
<td>01</td>
<td>00000511</td>
<td>CV</td>
<td>CSS</td>
<td>CV condition signal success</td>
</tr>
<tr>
<td>01</td>
<td>00000512</td>
<td>CV</td>
<td>CSM</td>
<td>CV condition signal miss</td>
</tr>
<tr>
<td>01</td>
<td>00000513</td>
<td>CV</td>
<td>CBS</td>
<td>CV condition broadcast success</td>
</tr>
<tr>
<td>01</td>
<td>00000514</td>
<td>CV</td>
<td>CBM</td>
<td>CV condition broadcast miss</td>
</tr>
<tr>
<td>01</td>
<td>00000515</td>
<td>CV</td>
<td>IN</td>
<td>CV initialize</td>
</tr>
<tr>
<td>01</td>
<td>00000516</td>
<td>CV</td>
<td>DS</td>
<td>CV destroy</td>
</tr>
<tr>
<td>01</td>
<td>00000522</td>
<td>CV</td>
<td>CIN</td>
<td>Call to SMC_INIT</td>
</tr>
<tr>
<td>01</td>
<td>00000523</td>
<td>CV</td>
<td>CSD</td>
<td>Call to SMC_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>00000529</td>
<td>CV</td>
<td>CSP</td>
<td>Call to SMC_POST</td>
</tr>
</tbody>
</table>
### Table 32. LE=2 entry records (continued)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0000052A</td>
<td>CV</td>
<td>CSB</td>
<td>Call to SMC_POSTALL</td>
</tr>
<tr>
<td>01</td>
<td>0000052B</td>
<td>CV</td>
<td>CSW</td>
<td>Call to SMC_WAIT</td>
</tr>
<tr>
<td>01</td>
<td>0000052C</td>
<td>CV</td>
<td>DBM</td>
<td>Shared condition broadcast - miss</td>
</tr>
<tr>
<td>01</td>
<td>0000052D</td>
<td>CV</td>
<td>DBS</td>
<td>Shared condition broadcast - success</td>
</tr>
<tr>
<td>01</td>
<td>0000052E</td>
<td>CV</td>
<td>DDS</td>
<td>Destroy (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>0000052F</td>
<td>CV</td>
<td>DIN</td>
<td>Initialize (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000530</td>
<td>CV</td>
<td>DSM</td>
<td>Condition signal - miss (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000531</td>
<td>CV</td>
<td>DSS</td>
<td>Condition signal - success (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000532</td>
<td>CV</td>
<td>DWI</td>
<td>Wait interrupted (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000533</td>
<td>CV</td>
<td>DTO</td>
<td>Wait timeout (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000534</td>
<td>CV</td>
<td>DWP</td>
<td>Wait posted (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006CB</td>
<td>CE</td>
<td>FBT</td>
<td>Error - Invalid system TOD (shared)</td>
</tr>
<tr>
<td>01</td>
<td>000006D1</td>
<td>CE</td>
<td>FRM</td>
<td>Error - Recursive mutex (shared)</td>
</tr>
<tr>
<td>01</td>
<td>000006D2</td>
<td>CE</td>
<td>FUO</td>
<td>Error - Shared mutex unowned</td>
</tr>
<tr>
<td>01</td>
<td>000006D3</td>
<td>CE</td>
<td>FDB</td>
<td>Error - Destroy failed (busy) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D4</td>
<td>CE</td>
<td>FDU</td>
<td>Error - Destroy failed (uninitialized) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D5</td>
<td>CE</td>
<td>FP</td>
<td>Error - Program check (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D6</td>
<td>CE</td>
<td>FUI</td>
<td>Error - Shared mutex or CV uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000006D7</td>
<td>CE</td>
<td>ENV</td>
<td>Error - BPX1SMC(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000006D8</td>
<td>CE</td>
<td>EPE</td>
<td>Error - BPX1SMC(fail) returns EPERM</td>
</tr>
<tr>
<td>01</td>
<td>000006D9</td>
<td>CE</td>
<td>EAN</td>
<td>Error - BPX1SMC(fail) returns EAGAIN</td>
</tr>
<tr>
<td>01</td>
<td>000006DA</td>
<td>CE</td>
<td>EIB</td>
<td>Error - BPX1SMC failed (EBUSY)</td>
</tr>
<tr>
<td>01</td>
<td>000006DB</td>
<td>CE</td>
<td>ESC</td>
<td>Error - BPX1SMC failed</td>
</tr>
<tr>
<td>01</td>
<td>000006EB</td>
<td>CE</td>
<td>EDD</td>
<td>CV destroy failed (damage)</td>
</tr>
<tr>
<td>01</td>
<td>000006EC</td>
<td>CE</td>
<td>EDB</td>
<td>CV destroy failed (busy)</td>
</tr>
<tr>
<td>01</td>
<td>000006ED</td>
<td>CE</td>
<td>EIA</td>
<td>CV initialization failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000006EE</td>
<td>CE</td>
<td>EIS</td>
<td>CV initialization failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000006EF</td>
<td>CE</td>
<td>EF</td>
<td>CV forced by quiesce</td>
</tr>
<tr>
<td>01</td>
<td>000006F0</td>
<td>CE</td>
<td>EP</td>
<td>CV program check</td>
</tr>
<tr>
<td>01</td>
<td>000006F1</td>
<td>CE</td>
<td>EBT</td>
<td>CV invalid system TOD</td>
</tr>
<tr>
<td>01</td>
<td>000006F2</td>
<td>CE</td>
<td>EBN</td>
<td>CV invalid timespec (nanoseconds)</td>
</tr>
<tr>
<td>01</td>
<td>000006F3</td>
<td>CE</td>
<td>EBS</td>
<td>CV invalid timespec (seconds)</td>
</tr>
<tr>
<td>01</td>
<td>000006F4</td>
<td>CE</td>
<td>EPO</td>
<td>CV condition post callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F5</td>
<td>CE</td>
<td>ETW</td>
<td>CV condition timed wait callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F6</td>
<td>CE</td>
<td>EWA</td>
<td>CV condition wait callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F7</td>
<td>CE</td>
<td>ESE</td>
<td>CV condition setup callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F8</td>
<td>CE</td>
<td>ERM</td>
<td>CV recursive mutex</td>
</tr>
<tr>
<td>01</td>
<td>000006F9</td>
<td>CE</td>
<td>EWM</td>
<td>CV wrong mutex</td>
</tr>
<tr>
<td>Member ID</td>
<td>Record Type</td>
<td>Class</td>
<td>Event</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>-------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>01</td>
<td>000006FA</td>
<td>CE</td>
<td>EDU</td>
<td>CV destroy failed (uninitialized)</td>
</tr>
<tr>
<td>01</td>
<td>000006FB</td>
<td>CE</td>
<td>EUI</td>
<td>CV mutex or CV uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000006FC</td>
<td>CE</td>
<td>EUO</td>
<td>CV mutex unowned</td>
</tr>
<tr>
<td>01</td>
<td>000006FE</td>
<td>CE</td>
<td>EIN</td>
<td>CV initialization failed (duplicate)</td>
</tr>
<tr>
<td>01</td>
<td>00000702</td>
<td>RW</td>
<td>R</td>
<td>Release</td>
</tr>
<tr>
<td>01</td>
<td>00000704</td>
<td>RW</td>
<td>AW</td>
<td>Acquire after wait</td>
</tr>
<tr>
<td>01</td>
<td>00000706</td>
<td>RW</td>
<td>I</td>
<td>Increment (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000707</td>
<td>RW</td>
<td>D</td>
<td>Decrement (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000715</td>
<td>RW</td>
<td>IN</td>
<td>Initialize</td>
</tr>
<tr>
<td>01</td>
<td>00000716</td>
<td>RW</td>
<td>DS</td>
<td>Destroy</td>
</tr>
<tr>
<td>01</td>
<td>00000717</td>
<td>RW</td>
<td>RA</td>
<td>Read acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000718</td>
<td>RW</td>
<td>WA</td>
<td>Write acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000719</td>
<td>RW</td>
<td>RB</td>
<td>Read busy (tryread failed)</td>
</tr>
<tr>
<td>01</td>
<td>0000071A</td>
<td>RW</td>
<td>WB</td>
<td>Write busy (trywrite failed)</td>
</tr>
<tr>
<td>01</td>
<td>0000071B</td>
<td>RW</td>
<td>RW</td>
<td>Read wait</td>
</tr>
<tr>
<td>01</td>
<td>0000071C</td>
<td>RW</td>
<td>WW</td>
<td>Write wait</td>
</tr>
<tr>
<td>01</td>
<td>0000071D</td>
<td>RW</td>
<td>BI</td>
<td>Call to SLK_INIT</td>
</tr>
<tr>
<td>01</td>
<td>0000071E</td>
<td>RW</td>
<td>BD</td>
<td>Call to SLK_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>0000071F</td>
<td>RW</td>
<td>BO</td>
<td>Call to SLK_OBTAIN</td>
</tr>
<tr>
<td>01</td>
<td>00000720</td>
<td>RW</td>
<td>BC</td>
<td>Call to SLK_OBTAIN_COND</td>
</tr>
<tr>
<td>01</td>
<td>00000721</td>
<td>RW</td>
<td>BR</td>
<td>Call to SLK_RELEASE</td>
</tr>
<tr>
<td>01</td>
<td>000008DC</td>
<td>RE</td>
<td>EOW</td>
<td>Error - Already owned for write (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008DD</td>
<td>RE</td>
<td>EOR</td>
<td>Error - Already owned for read (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008DE</td>
<td>RE</td>
<td>EDL</td>
<td>Error - BPX1SLK(fail) returns EDEADLK</td>
</tr>
<tr>
<td>01</td>
<td>000008DF</td>
<td>RE</td>
<td>EIV</td>
<td>Error - BPX1SLK(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000008E0</td>
<td>RE</td>
<td>EPM</td>
<td>Error - BPX1SLK(fail) returns EPERM</td>
</tr>
<tr>
<td>01</td>
<td>000008E1</td>
<td>RE</td>
<td>EAG</td>
<td>Error - BPX1SLK(fail) returns EAGAIN</td>
</tr>
<tr>
<td>01</td>
<td>000008E2</td>
<td>RE</td>
<td>EBS</td>
<td>Error - BPX1SLK(fail) returns EBUSY</td>
</tr>
<tr>
<td>01</td>
<td>000008E3</td>
<td>RE</td>
<td>ENM</td>
<td>Error - BPX1SLK(fail) returns ENOMEM</td>
</tr>
<tr>
<td>01</td>
<td>000008E4</td>
<td>RE</td>
<td>EBR</td>
<td>Error - BPX1SLK(RELEASE) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E5</td>
<td>RE</td>
<td>EBC</td>
<td>Error - BPX1SLK(OBTAIN_COND) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E6</td>
<td>RE</td>
<td>EBO</td>
<td>Error - BPX1SLK(OBTAIN) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E7</td>
<td>RE</td>
<td>EBD</td>
<td>Error - BPX1SLK(DESTROY) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E8</td>
<td>RE</td>
<td>EBI</td>
<td>Error - BPX1SLK(INIT) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E9</td>
<td>RE</td>
<td>EFR</td>
<td>Error - Forced release</td>
</tr>
<tr>
<td>01</td>
<td>000008EA</td>
<td>RE</td>
<td>EFD</td>
<td>Error - Forced decrement</td>
</tr>
<tr>
<td>01</td>
<td>000008ED</td>
<td>RE</td>
<td>EIA</td>
<td>Error - Initialization failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000008EE</td>
<td>RE</td>
<td>EIS</td>
<td>Error - Initialization failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000008EF</td>
<td>RE</td>
<td>EF</td>
<td>Error - Forced by quiesce</td>
</tr>
</tbody>
</table>
Table 32. LE=2 entry records (continued)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>000008F0</td>
<td>RE</td>
<td>EP</td>
<td>Error - Program check</td>
</tr>
<tr>
<td>01</td>
<td>000008FB</td>
<td>RE</td>
<td>EU</td>
<td>Error - Uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000008FC</td>
<td>RE</td>
<td>EUO</td>
<td>Error - Unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000008FD</td>
<td>RE</td>
<td>EO</td>
<td>Error - Already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008FE</td>
<td>RE</td>
<td>EIN</td>
<td>Error - Initialization failed (duplicate)</td>
</tr>
</tbody>
</table>

Table 33 shows the format for the Mutex – Condition Variable – Latch entries in the trace table.

Table 33. Format of the mutex/CV/latch records

<table>
<thead>
<tr>
<th>Record fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
</tr>
<tr>
<td>unused</td>
</tr>
</tbody>
</table>

Class  Two character EBCDIC representation of the trace class.
- LT  Latch
- LE  Latch Exception
- MX  Mutex
- ME  Mutex Exception
- CV  Condition Variable
- CE  Condition Variable Exception

Source  One character EBCDIC representation of the event.
- C  C/C++
- S  Sockets

Blank  Blank character

Event  Two character EBCDIC representation of the event. See Table 32 on page 162.

Object Addr  Fullword address of the mutex object.

Name 1  Optional eight character field containing the name of the function or object to be recorded.

Name 2  Optional eight character field containing the name of the function or object to be recorded.

When LE=3 is specified: The trace table will include the records generated by both LE=1 and LE=2.

When LE=8 is specified: The trace table will contain only storage allocation records, as shown in Table 34 on page 168. Currently this is only supported by C/C++. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 209.
Table 34. LE=8 entry records

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000001</td>
<td>Storage allocation entry</td>
</tr>
<tr>
<td>03</td>
<td>00000001</td>
<td>Storage allocation exit</td>
</tr>
</tbody>
</table>

When LE=20 is specified: Table 35 shows the C/C++ records that might be generated. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 209.

Table 35. LE=20 entry records

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000007</td>
<td>XPLINK calls non-XPLINK entry</td>
</tr>
<tr>
<td>03</td>
<td>00000008</td>
<td>non-XPLINK calls XPLINK entry</td>
</tr>
</tbody>
</table>

Sample dump for the trace table entry

The following sample shows an example of a dump of the trace table when you specify the LE=1 suboption (the library call/return trace).

```
<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000007</td>
<td>XPLINK calls non-XPLINK entry</td>
</tr>
<tr>
<td>03</td>
<td>00000008</td>
<td>non-XPLINK calls XPLINK entry</td>
</tr>
</tbody>
</table>
```
Requesting a UNIX System Services syscall trace for debugging

Signal SIGTRACE can be sent to a process or process group to start or stop a trace of the z/OS UNIX System Services syscalls made by the application. The signal is implemented as a toggle. With the trace turned on, the z/OS UNIX System Services kernel gathers the syscall trace records for the targeted processes. A system dump of the user address space can be generated by sending signal SIGDUMP to the same processes to capture the trace output. See z/OS UNIX System Services Command Reference for more information about the SIGTRACE signal.
Part 2. Debugging language-specific routines

This part provides specific information for debugging applications written in C/C++, COBOL, Fortran, and PL/I. It also discusses techniques for debugging under CICS.
Chapter 4. Debugging C/C++ routines

This chapter provides specific information to help you debug applications that contain one or more C/C++ routines. It also provides information about debugging C/C++ applications compiled with XPLINK. It includes the following topics:

- Debugging C/C++ I/O routines
- Using C/C++ compiler listings
- Generating a Language Environment dump of a C/C++ routine
- Generating a Language Environment dump of a C/C++ routine with XPLINK
- Finding C/C++ information in a Language Environment dump
- Debugging example of C/C++ routines
- Debugging example of C/C++ routines with XPLINK

There are several debugging features that are unique to C/C++ routines. Before examining the C/C++ techniques to find errors, you might want to consider the following areas of potential problems:

- If you suspect that you are using uninitialized storage, you may want to use the STORAGE runtime option.
- If you are using the fetch() function, see z/OS XL C/C++ Programming Guide to ensure that you are creating the fetchable module correctly.
- If you are using DLLs, see z/OS XL C/C++ Programming Guide to ensure that you are using the DLL correctly.
- For non-System Programming C routines, ensure that the entry point of the load module is CEESTART.
- You should avoid:
  - Incorrect casting
  - Referencing an array element with a subscript outside the declared bounds
  - Copying a string to a target with a shorter length than the source string
  - Declaring but not initializing a pointer variable, or using a pointer to allocated storage that has already been freed

If a routine exception occurred and you need more information than the condition handler provided, run your routine with the following runtime options: TRAP(ON, NOSPIE) and TERMTHDACT(UAIMM). Setting these runtime options generates a system dump of the user address space of the original abend or program interrupt prior to the Language Environment condition manager processing the condition. After the system dump is taken by the operating system, the Language Environment condition manager continues processing.

Debugging C/C++ programs

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

Using the __amrc and __amrc2 structures to debug input/output

__amrc, a structure defined in stdio.h, can help you determine the cause of errors resulting from an I/O operation, because it contains diagnostic information (for example, the return code from a failed VSAM operation). There are two structures:
• __amrc (defined by type __amrc_type
• __amrc2 (defined by type __amrc2_type)

The __amrc2_type structure contains secondary information that C can provide.

Because any I/O function calls, such as printf(), can change the value of __amrc or __amrc2, make sure you save the contents into temporary structures of __amrc_type and __amrc2_type respectively, before dumping them.

Figure 36 shows the structure as it appears in stdio.h.

```c
typedef struct __amrctype {
    union {
        long int __error;
        struct {
            unsigned short __syscode,
            __rc;
        } __abend;
        struct {
            unsigned char __fdbk_fill,
            __rc,
            __ftncd,
            __fdbk;
        } __feedback;
        struct {
            unsigned short __svc99_info,
            __svc99_error;
        } __alloc;
        } __code;
        unsigned long __RBA;
        unsigned int __last_op;
        struct {
            unsigned long __len_fill; /* __len + 4 */
            unsigned long __len;
            char __str1[120];
            unsigned long __parmr0;
            unsigned long __parmr1;
            unsigned long __fill2[2];
            char __str2[64];
        } __msg;
        #if __EDC_TARGET >= 0x22080000
        unsigned char __rplfdbwd[4];
        #endif
        #if __EDC_TARGET >= 0x41080000
        #ifdef __LP64
        unsigned long __XRBA;
        #elif defined(__LL)
        unsigned long __XRBA;
        #else
        unsigned int __XRBA1;
        unsigned int __XRBA2;
        #endif
        unsigned char __amrc_noseek_to_seek;
        char __amrc_pad[23];
        #endif
    } __amrc_type;
```
Figure 37 shows the __amrc2 structure as it appears in stdio.h.

```
struct {
    long int __error2;
    char __pad_error2[4];
    FILE *__fileptr;
    long int __reserved{6};
}
```

**Figure 37. __amrc2 structure**

1. `union { ... } __code`
   The error or warning value from an I/O operation is in __error, __abend, __feedback, or __alloc. Look at __last_op to determine how to interpret the __code union.

2. `__error`
   A structure that contains error codes for certain macros or services your application uses. Look at __last_op to determine the error codes. __syscode is the system abend code.

3. `__abend`
   A structure that contains the abend code when errno is set to indicate a recoverable I/O abend. __rc is the return code. For more information on abend codes, see z/OS MVS System Codes.

4. `__feedback`
   A structure that is used for VSAM only. The __rc stores the VSAM register 15, __fdbk stores the VSAM error code or reason code, and __RBA stores the RBA after some operations.

5. `__alloc`
   A structure that contains errors during fopen or freopen calls when defining files to the system using SVC 99.

6. `__RBA`
   The RBA value returned by VSAM after an ESDS or KSDS record is written out. For an RRDS, it is the calculated value from the record number. It can be used in subsequent calls to flocate.

7. `__last_op`
   A field containing a value that indicates the last I/O operation being performed by C/C++ at the time the error occurred. These values are shown in Table 36 on page 176.

8. `__msg`
   May contain the system error messages from read or write operations emitted from the DFSMS/MVS SYNADAF macro instruction. Because the message can start with a hexadecimal address followed by a short integer, it is advisable to start printing at MSG+6 or greater so the message can be printed as a string. Because the message is not null-terminated, a maximum of 114 characters should be printed. This can be accomplished by specifying a printf format specifier as %.114s.

9. `__amrc_noseek_to_seek`
   This field contains the reason for the switch from QSAM (noseek) to BSAM with NOTE and POINT macros requested (seek) by the XL C/C++ Runtime...
Library. This field is set when system-level I/O macro processing triggers an ABEND condition. The macro name values (defined in stdio.h) for this field are as follows:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>__AM_BSAM_NOSWITCH</td>
<td>No switch was made.</td>
</tr>
<tr>
<td>__AM_BSAM_UPDATE</td>
<td>The data set is open for update</td>
</tr>
<tr>
<td>__AM_BSAM_BSAMWRITE</td>
<td>The data set is already open for write (or update) in the same C process.</td>
</tr>
<tr>
<td>__AM_BSAM_FBS_APPEND</td>
<td>The data set is recfm=FBS and open for append</td>
</tr>
<tr>
<td>__AM_BSAM_LRECLX</td>
<td>The data set is recfm=LRECLX (used for VBS data sets where records span the largest blocksize allowed on the device)</td>
</tr>
<tr>
<td>__AM_BSAM_PARTITIONED_DIRECTORY</td>
<td>The data set is the directory for a regular or extended partitioned data set</td>
</tr>
<tr>
<td>__AM_BSAM_PARTITIONED INDIRECT</td>
<td>The data set is a member of a partitioned data set, and the member name was not specified at allocation</td>
</tr>
</tbody>
</table>

[10] __XRBA
This is the 8 byte relative byte address returned by VSAM after an EDS or KSDS record is written out. For an RRDS, it is the calculated value from the record number. It may be used in subsequent calls to flocate().

A secondary error code. For example, an unsuccessful rename or remove operation places its reason code here.

[12] __fileptr
A pointer to the file that caused a SIGIOERR to be raised. Use an fldata() call to get the actual name of the file.

[13] __reserved
Reserved for future use.

__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 36 lists __last_op values you could receive and where to look for further information.

Table 36: __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>
</tbody>
</table>
Table 36. __last_op values and diagnosis information (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<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_BLDL</td>
<td>Sets __error with return code from OS BLDL 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>__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</td>
</tr>
<tr>
<td></td>
<td>with no newline such that the record fills up to capacity and subsequent characters</td>
</tr>
<tr>
<td></td>
<td>cannot be written. For a record I/O file this refers to an fwrite() writing more</td>
</tr>
<tr>
<td></td>
<td>data than the record can hold. Truncation is always rightmost data. There is no</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>physical record for anymore double byte characters. A new-line is not acceptable at</td>
</tr>
<tr>
<td></td>
<td>this point. Truncation will continue to occur until an SI is written or the file</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>else when a redundant SO is written to the file before an SI. Cannot happen if MB_C</td>
</tr>
<tr>
<td></td>
<td>UR_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</td>
</tr>
<tr>
<td></td>
<td>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,</td>
</tr>
<tr>
<td></td>
<td>thereby forcing C or C++ to fill in the last byte of the DBCS string with a padding</td>
</tr>
<tr>
<td></td>
<td>byte X'FE'. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------</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 RRDS.</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 KSDS.</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 PATH.</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 KSDS PATH.</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>
<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>__CMS_OPEN</td>
<td>Sets _error to result of OS OPEN macro.</td>
</tr>
<tr>
<td>__CMS_CLOSE</td>
<td>Sets _error to result of OS CLOSE 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>
</tbody>
</table>
Table 36: __last_op values and diagnosis information  (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__CMS_READ</td>
<td>Sets __err error to result of FSREAD.</td>
</tr>
<tr>
<td>__CMS_WRITE</td>
<td>Sets __err error to result of FSWRITE.</td>
</tr>
<tr>
<td>__CMS_STATE</td>
<td>Sets __err error to result of FSSTATE.</td>
</tr>
<tr>
<td>__CMS_ERASE</td>
<td>Sets __err error to result of FSERASE.</td>
</tr>
<tr>
<td>__CMS_RENAME</td>
<td>Sets __err error to result of CMS RENAME command.</td>
</tr>
<tr>
<td>__CMS_EXTRACT</td>
<td>Sets __err error to result of DMS EXTRACT call.</td>
</tr>
<tr>
<td>__CMS_LINERD</td>
<td>Sets __err error to result of LINERD macro.</td>
</tr>
<tr>
<td>__CMS_LINEWRT</td>
<td>Sets __err error to result of LINEWRT macro.</td>
</tr>
<tr>
<td>__CMS_QUERY</td>
<td>__err is not set.</td>
</tr>
<tr>
<td>__HSP_CREATE</td>
<td>Indicates last op was a DSPSERV CREATE to create 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>__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 __err error with error code from EXEC CICS WRITEQ TD.</td>
</tr>
<tr>
<td>__LFS_OPEN</td>
<td>Sets __err 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: Assembly Callable Services Reference</td>
</tr>
<tr>
<td>__LFS_CLOSE</td>
<td>Sets __err 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: Assembly Callable Services Reference</td>
</tr>
<tr>
<td>__LFS_READ</td>
<td>Sets __err 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: Assembly Callable Services Reference</td>
</tr>
<tr>
<td>__LFS_WRITE</td>
<td>Sets __err 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: Assembly Callable Services Reference</td>
</tr>
</tbody>
</table>
Table 36. __last_op values and diagnosis information (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<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: <a href="#">Assembler Callable Services Reference</a></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: <a href="#">Assembler Callable Services Reference</a></td>
</tr>
</tbody>
</table>

Using file I/O tracing to debug C/C++ file I/O problems

You can use file I/O tracing to debug C/C++ file I/O problems. For more information, see [Debugging I/O programs in z/OS XL C/C++ Programming Guide](#).

Displaying an error message with the perror() function

To find a failing routine, check the return code of all function calls. After you have found the failing routine, use the perror() function after the routine to display the error message. perror() displays the string that you pass to it and an error message corresponding to the value of errno. perror() writes to the standard error stream (stderr). Figure 38 is an example of a routine using perror().

By default, the errno2 value will be appended to the end of the perror() string. If you do not want the errno2 value appended to the perror() string, set the _EDC_ADD_ERRNO2 environment variable to 0.

```c
#include <stdio.h>
int main(void){
    FILE *fp;
    fp = fopen("myfile.dat", "w");
    if (fp == NULL)
        perror("fopen error");
}
```

Figure 38. Example of a routine using perror()

Using __errno2() to diagnose application problems

Use the __errno2() function when diagnosing problems in an application program. This function enables z/OS XL C/C++ application programs to access additional diagnostic information, errno2 (errnojr), associated with errno. The __errno2 may be set by the z/OS XL C/C++ runtime library, z/OS UNIX callable services, or other callable services. The errno2 is intended for diagnostic display purposes only and is not a programming interface.

**Note:** Not all functions set errno2 when errno is set. In the cases where errno2 is not set, the __errno2() function may return a residual value. You may use the __err2ad() function to clear errno2 to reduce the possibility of a residual value being returned.

Figure 39 on page 181 is an example of a routine using __errno2().
Figure 39. Example of a routine using __errno2()

Figure 40 shows the output from the sample routine in Figure 39.

```c
#pragma runopts(posix(on))
#define _EXT
#include <stdio.h>
#include <errno.h>

int main(void) {
    FILE *f;
    f = fopen("testfile.dat", "r");
    if (f==NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
        return 0;
    }
}
```

Figure 40. Sample output of a routine using __errno2()

fopen() failed: EDCS1291 No such file or directory. (errno2=0x05620062)
__errno2 = 05620062

Figure 41. Example of a routine using _EDC_ADD_ERRNO2

```c
#pragma runopts(posix(on))
#define _EXT
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(void) {
    FILE *fp;
    /* do NOT add errno2 to perror message */
    setenv("_EDC_ADD_ERRNO2", "0", 1);
    fp = fopen("testfile.dat", "r");
    if (fp == NULL)
        perror("fopen() failed");
    return 0;
}
```

Figure 41. Example of a routine using _EDC_ADD_ERRNO2

Figure 42 shows the sample output from the routine in Figure 41.

fopen() failed: EDCS1291 No such file or directory.

Figure 42. Sample output of a routine using _EDC_ADD_ERRNO2

Figure 43 on page 182 is an example of a routine using __err2ad() in combination with __errno2().
```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(void) {
    FILE *f;
    setenv("_EDC_ADD_ERRNO2", "0", 1);
    f = fopen("testfile.dat", "r");
    if (f == NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x, __errno2()");
    } /* reset errno2 to zero */
    __errno2 = 0x0;
    printf("__errno2 = %08x", __errno2());
    f = fopen("testfile.dat", "r");
    if (f == NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x", __errno2());
    }
    return 0;
}
```

Figure 43. Example of a routine using __err2ad() in combination with __errno2()

Figure 44 shows the sample output from the routine shown in Figure 43

fopen() failed: EDC5129I No such file or directory.
__errno2 = 05620062
__errno2 = 00000000
fopen() failed: EDC5129I No such file or directory.
__errno2 = 05620062

Figure 44. Sample output of routine using __err2ad() in combination with __errno2()

For more information about _EDC_ADD_ERRNO2, see z/OS XL C/C++ Programming Guide

For more information about __errno2() and __err2ad(), see z/OS XL C/C++ Runtime Library Reference

**Diagnosing DLL problems**

Use the _EDC_DLL_DIAG environment variable to diagnose DLL problems. For more information, see z/OS XL C/C++ Programming Guide

You can also see the diagnosis output in CEEDUMP and Verbexit LEDATA reports. For more information, see “Using the DLL failure control block” on page 82

**Using C/C++ listings**

For a detailed description of available listings, see z/OS XL C/C++ User’s Guide
Finding variables

You can determine the value of a variable in the routine at the point of interrupt by using the compiled code listing as a guide to its address, then finding this address in the Language Environment dump. The method you use depends on the storage class of variable.

This method is generally used when no symbolic variables have been dumped (by using the TEST compiler option).

It is possible for the routine to be interrupted before the value of the variable is placed in the location provided for it. This can explain unexpected values in the dump.

Steps for finding automatic variables

Perform the following steps to find automatic variables in the Language Environment dump:

1. Identify the start of the stack frame. If a dump has been taken, each stack frame is dumped. The stack frames can be cross-referenced to the function name in the traceback.

2. Determine the value of the base register (in this example, GPR13) in the Saved Registers section for the function you are interested in.

3. Find the offset of the variable (which is given in decimal) in the storage offset listing.

   aal 85-0:85  Class = automatic, Offset = 164(r13),  Length = 40

4. Add this base address to the offset of the variable.

When you are done, the contents of the variable can be read in the DSA Frame section corresponding to the function the variable is contained in.

Locating the Writable Static Area (WSA)

The Writable Static Area (WSA) address is the base address of the writable static area which is available for all C and C++ programs except C programs compiled with the NORENT compiler option. If you have C code compiled with the RENT option or C++ code (hereafter called RENT code) you must determine the base address of the WSA if you want to calculate the address of a static or external variable. Use the following table to determine where to find the WSA base address:

<table>
<thead>
<tr>
<th>Table 37. Finding the WSA base address</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you want the WSA base address for:</td>
</tr>
<tr>
<td>application code</td>
</tr>
<tr>
<td>a fetched module</td>
</tr>
<tr>
<td>a DLL</td>
</tr>
<tr>
<td>Locate the WSA base address in:</td>
</tr>
<tr>
<td>the WSA address field in the Enclave Control Blocks section</td>
</tr>
<tr>
<td>the WSA address field of the Fetch() Information section for the fetch() function pointer for which you are interested</td>
</tr>
<tr>
<td>the corresponding WSA address in the DLL Information section</td>
</tr>
</tbody>
</table>

Use the WSA base address to locate the WSA in the Enclave Storage section.

Steps for finding the static storage area

If you have C code compiled with the NORENT option (hereafter called NORENT code) you must determine the base address of the static storage area if you want to calculate the address of a static or external variable.
Perform the following steps to find the static storage area:

1. Name the static storage area CSECT by using the \texttt{pragma csect} directive. Once this is done, a CSECT is generated for the static storage area for each source file.
2. Determine the origin and length of the CSECT from the linker map.
3. Locate the external variables corresponding to the CSECT with the same name.
4. Determine the origin and length of the external variable CSECT from the linker map.

\textbf{Note:}

1. Address calculation for static and external variables uses the static storage area as a base address with 1 or more offsets added to this address.
2. The storage associated with these CSECTs is not dumped when an exception occurs. It is dumped when \texttt{cdump} or CEE3DMP is called, but it is written to a separate ddname called CEESNAP. For information about \texttt{cdump}, CEE3DMP, and enabling the CEESNAP ddname, see "Generating a Language Environment dump of a C/C++ routine" on page 190.

\textbf{Steps for finding RENT static variables}

Before you begin: you need to know the WSA. To find this information, see "Locating the Writable Static Area (WSA)" on page 183. For this procedure's example, assume that the address of writable static is X'02D66E40'.

Perform the following steps to find RENT static variables:

1. Find the offset of @\texttt{STATIC} (associated with the file where the static variable is located) in the Writable Static Map section of the prelinker map. Figure 45 on page 185 shows an example; in this Writable Static Map section of a prelinker map, the offset is X'58'.
2. Add the offset to the WSA to get the base address of static variables, as shown.

\[ X'02D66E40' + X'58' = X'02D66E98' \]

3. Find the offset of the static variable in the partial storage offset compiler listing.
   In the following example, the offset is 96 (X'60').
   sa0 66-0:66 Class = static, Location = WSA + @STATIC + 96, Length = 4

4. Add the offset of the static variable in the partial storage offset compiler listing
   (found in step 3) to the base address of static variables (calculated in step 2).
   \[ X'02D66E98' + X'60' = X'02D66EF8' \]

When you are done, you have the address of the value of the static variable in the
Language Environment dump.

Figure 46 shows the path to locate RENT C++ and C static variables by adding the
address of writable static, the offset of @STATIC, and the variable offset.

---

**Steps for finding external RENT variables**

**Before you begin:** You need to know the WSA. To find this information see
"Locating the Writable Static Area (WSA)" on page 183. For this procedure's
example, the address of writable static is X'02D66E40'.
Perform the following steps to find external RENT variables:

1. Find the offset of the external variable in the Prelinker Writable Static Map. In the example shown in Figure 47, the offset for DFHEIPTR is X'28'.

<table>
<thead>
<tr>
<th>Writable Static Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>1C</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>2C</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>3C</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>44</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>4C</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>54</td>
</tr>
<tr>
<td>58</td>
</tr>
</tbody>
</table>

Figure 47. Writable static map produced by prelinker

2. Add the offset of the external variable to the address of writable static, as shown below.

   X'02D66E40' + X'28' = X'2D66E68'

When you are done, you have the address of the value of the external variable in the Language Environment dump.

Steps for finding NORENT static variables

Before you begin: You need to know the name and address of the static storage area. To find this information see Steps for finding the static storage area on page 183. For this procedure's example, the static storage area is called STATSTOR and has an address of X'02D66E40'.

Perform the following steps to find external RENT variables:

1. Find the offset of the static variable in the partial storage offset compiler listing. As shown in the following example, the offset is 96 (X'60').

   sa0  66-0:66 Class = static, Location = STATSTOR +96, Length = 4

2. Add the offset to the base address of static variables, as shown in the following example:

   X'2D66E40' + X'60' = X'2D66EA0'

When you are done, you have the address of the value of the static variable in the Language Environment dump.
Figure 48 shows how to locate NORENT C static variables by adding the Static Storage Area CSECT address to the variable offset.

Steps for finding external NORENT variables

Before you begin: You need to find the address of the external variable CSECT. To find this information, see “Steps for finding the static storage area” on page 183. For this procedure's example, the address of the external variable CSECT is X'02D66E40'.

The address of the external variable CSECT is the address of the value of the external variable in the Language Environment dump.

Steps for finding the C/370 parameter list

Perform the following steps to locate a parameter in the Language Environment dump:

1. Identify the address of the start of the parameter list. A pointer to the parameter list is passed to the called function in register 1. This is the address of the start of the parameter list. Figure 49 shows an example code for the parameter variable.

```
func0() {
  ...
  func1(a1,a2);
  ...
}

func1(int ppx, int pp0) {
  ...
}
```

Figure 49. Example code for parameter variable

Parameters ppx and pp0 correspond to copies of a1 and a2 in the stack frame belonging to func0.

2. Use the address of the start of the parameter list to find the register and offset in the partial storage offset listing. As shown in the following example, the offset is 4 (X'4') from register 1.

```
pp0  62-0:62  Class = parameter,  Location = 4(r1),  Length = 4
```

3. Determine the value of GPR1 in the Saved Registers section for the function that called the function you are interested in.

4. Add this base address to the offset of the parameter.

When you are done, the contents of the variable can then be read in the DSA frame section corresponding to the function the parameter was passed from.
Steps for finding the C++ parameter list

Before you begin: To locate C++ functions with extern C attributes, see "Steps for finding the C/370 parameter list" on page 187.

Perform the following steps to find the C++ parameter list:

1. Identify the address of the start of the parameter list. A pointer to the parameter list is passed to the called function in register 1. This is the address of the start of the parameter list. Figure 50 shows an example code for the parameter variable.

```c
func0() {
  func1(a1,a2);
}
func1(int ppx, int pp0) {
  
}
```

Figure 50. Example code for parameter variable

Parameters ppx and pp0 correspond to copies of a1 and a2 in the stack frame belonging to func1.

2. Locate the value of the base register in the Saved Registers section of the function you are interested in.

3. Find the offset of the static variable in the partial storage offset compiler listing, as shown in Figure 51.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Class</th>
<th>Location</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppx</td>
<td>parameter</td>
<td>188(r13)</td>
<td>4</td>
</tr>
<tr>
<td>pp0</td>
<td>parameter</td>
<td>192(r13)</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 51. Partial storage offset listing

4. Add the value of the base register to the offset.

5. Locate the parameter.

Restriction: When OPTIMIZE is on, the parameter value might never be stored, since the first few parameters might be passed in registers and there might be no need to save them.

Steps for finding members of aggregates

You can define aggregates in any of the storage classes or pass them as parameters to a called function. The first step is to find the start of the aggregate. You can compute the start of the aggregate as described in previous sections, depending on the type of aggregate used.

The aggregate map provided for each declaration in a routine can further assist in finding the offset of a specific variable within an aggregate. Structure maps are generated using the AGGREGATE compiler option. Figure 52 on page 189 shows an example of a static aggregate.
static struct {
    short int ss01;
    char ss02[56];
    int sz0[6];
    int ss03;
} ss0;

Figure 52. Example code for structure variable

Figure 53 shows an example aggregate map.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bytes(Bits)</th>
<th>Length</th>
<th>Bytes(Bits)</th>
<th>Member Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
<td>ss01</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>56</td>
<td></td>
<td>ss02[56]</td>
</tr>
<tr>
<td>58</td>
<td></td>
<td>2</td>
<td><em><strong>PADDING</strong></em></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>24</td>
<td></td>
<td>sz0[6]</td>
</tr>
<tr>
<td>84</td>
<td></td>
<td>4</td>
<td></td>
<td>ss03</td>
</tr>
</tbody>
</table>

Figure 53. Example of aggregate map

Assume the structure has been compiled as RENT. To find the value of variable sz0[0]:

1. Find the address of the writable static. For this example the address of writable static is X’02D66E40’.

2. Find the offset of @STATIC in the Writable Static Map. In this example, the offset is X’58’. Add this offset to the address of writable static. The result is X’2D66E98’ (X’02D66E40’ + X’58’), Figure 54 on page 190 shows the Writable Static Map produced by the prelinker.
3. Find the offset of the static variable in the storage offset listing. The offset is 96 (X’60’). The following is an example of a partial storage offset listing.

```
ss0 66-0:66 Class = static, Location = GPR13(96), Length = 4
Add this offset to the result from step 2. The result is X'2D66EF8' (X'2D66E98' + X'60'). This is the address of the value of the static variable in the dump.
```

4. Find the offset of sz0 in the Aggregate Map, shown in Figure 53 on page 189. The offset is 60.

Add the offset from the Aggregate Map to the address of the ss0 struct. The result is X'60' (X'3C' + X'60'). This is the address of the values of sz0 in the dump.

**Finding the timestamp**

The timestamp is in the compile unit block. The address for the compile unit block is located at eight bytes past the function entry point. The compile unit block is the same for all functions in the same compilation. The fourth word of the compile unit block points to the timestamp. The timestamp is 16 bytes long and has the following format:

```
YYYYMMDDHHMMSSSS
```

**Generating a Language Environment dump of a C/C++ routine**

You can use 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.

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 `//CEEDUMP DD` statement in MVS/JCL.
cdump(), csnap(), and ctrace() all return a 1 code in the SPC environment because they are not supported in SPC.

See the z/OS XL C/C++ Runtime Library Reference for more details about the syntax of these functions.

cdump()

If your routine is running under z/OS or CICS, 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.

The output of the dump is directed to the CEESNAP data set. The DD definition for CEESNAP is as follows:

```
//CEESNAP DD SYSOUT=* *
```

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 the following message, where nnn corresponds to the SNAP identifier described above. An unsuccessful SNAP does not result in an incrementation of the identifier.

```
Snap was successful; snap ID = nnn
```

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.

Support for SNAP dumps using the _cdump function is provided only under z/OS and z/VM. SNAP dumps are not supported under CICS; no SNAP is produced in this environment. A successful SNAP results in a large quantity of output. A routine calling cdump() under CICS receives a return code of 0 if the ensuing call to CEE3DMP is successful. In addition to a SNAP dump, a 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
```

ctrace()

The ctrace() function produces a traceback and includes the offset addresses from which the calls were made. ctrace() is equivalent to calling CEE3DMP with the option string:

```
TRACEBACK NOFILES NOBLOCKS NOVARIABLES NOSTORAGE STACKFRAME(ALL) NOCONDITION NOENTRY
```

Sample C routine that calls cdump()

The code example below shows a sample C routine that uses the cdump function to generate a dump. The sample here shows the dump output.

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

void hsigfpe(int);
void hsigterm(int);
void atf1(void);

typedef int (*FuncPtr_T)(void);

int st1 = 99;
int st2 = 255;
int xcount = 0;

int main(void) {
  /*
   * 1) Open multiple files
   * 2) Register 2 signals
   * 3) Register 1 atexit function
   * 4) Fetch and execute a module
   */

  FuncPtr_T fetchPtr;
  FILE *fp1;
  FILE *fp2;
  int rc;
  fp1 = fopen("myfile.data", "w");
  if (!fp1) {
    perror("Could not open myfile.data for write");
    exit(101);
  }
  fprintf(fp1, "record 1\n");
  fprintf(fp1, "record 2\n");
  fprintf(fp1, "record 3\n");

  fp2 = fopen("memory.data", "wb,type=memory");
  if (!fp2) {
    perror("Could not open memory.data for write");
    exit(102);
  }
  fprintf(fp2, "some data");
  fprintf(fp2, "some more data");
  fprintf(fp2, "even more data");
  signal(SIGFPE, hsigfpe);
```

192 z/OS Language Environment Debugging Guide
signal(SIGTERM, hsigterm);
rc = atexit(atf1);
if (rc) {
    fprintf(stderr, "Failed on registration of atexit function atf1\n");
exit(103);
}
fetchPtr = (FuncPtr_T) fetch("MODULE1");
if (!fetchPtr) {
    fprintf(stderr, "Failed to fetch MODULE1\n");
    exit(104);
}
fetchPtr();
return(0);
}

void hsigfpe(int sig) {
    ++st1;
    return;
}

void hsigterm(int sig) {
    ++st2;
    return;
}

void atf1() {
    ++xcount;
}

Figure 55 shows a fetched C module.

#include <ctest.h>
#pragma linkage(func1, fetchable)
int func1(void) {
    cdump("This is a sample dump");
    return(0);
}

Figure 55. Fetched module for C routine

Sample C++ routine that generates a Language Environment dump

Figure 56 on page 194 shows a sample C++ routine that uses a protection exception to generate a dump.
#include <iostream.h>
#include <ctest.h>
#include "stack.h"

int main() {
    cout << "Program starting:\n";
    cerr << "Error report:\n";
    Stack<int> x;
    x.push(1);
    cout << "Top value on stack: " << x.pop() << '\n';
    cout << "Next value on stack: " << x.pop() << '\n';
    return(0);
}

Figure 56. Example C++ routine with protection exception generating a dump

Figure 57 shows the template file stack.c

#ifndef __STACK__
#include "stack.h"
#endif

template <class T> T Stack<T>::pop() {
    T value = head->value;
    head = head->next;
    return(value);
}
template <class T> void Stack<T>::push(T value) {
    Node* newNode = new Node;
    newNode->value = value;
    newNode->next = head;
    head = newNode;
}

Figure 57. Template file STACK.C

Figure 58 shows the header file stack.h.

#ifndef __STACK__
#define __STACK__
template <class T> class Stack {
    public:
        Stack() { char* badPtr = 0; badPtr -= (0x01010101);
            head = (Node*) badPtr; /* head initialized to OxFEFEFEFF */
        } T pop();
        void push(T);
    private:
        struct Node {
            T value;
            struct Node* next;
        }* head;
};
#endif

Figure 58. Header file STACK.H
Sample Language Environment dump with C/C++-specific information

The sample dump below was produced by compiling the routine in this sample with the TEST(SYM) compiler option, then running it. Notice the sequence of calls in the traceback section - EDCZMINV is the C/C++ management module that invokes main and @FECBMODULE1 fetches the user-defined function func1, which in turn calls the library routine __cdump.

If source code is compiled with the GONUMBER or TEST compiler option, statement numbers are shown in the traceback. If source code is compiled with the TEST(SYM) compile option, variables and their associated type and value are dumped out. Note that the high half of register 14 at entry to CEE3DMP is not available and is shown in the dump as *******.

For more information about C/C++-specific information contained in a dump, see “Finding C/C++ information in a Language Environment dump” on page 199.

Registers on Entry to CEE3DMP:

<table>
<thead>
<tr>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>00000000</td>
<td>VR1</td>
<td>00000000</td>
<td>VR2</td>
<td>00000000</td>
<td>VR3</td>
<td>00000000</td>
<td>VR4</td>
<td>00000000</td>
<td>VR5</td>
<td>00000000</td>
<td>VR6</td>
<td>00000000</td>
<td>VR7</td>
<td>00000000</td>
</tr>
<tr>
<td>VR8</td>
<td>00000000</td>
<td>VR9</td>
<td>00000000</td>
<td>VR10</td>
<td>00000000</td>
<td>VR11</td>
<td>00000000</td>
<td>VR12</td>
<td>00000000</td>
<td>VR13</td>
<td>00000000</td>
<td>VR14</td>
<td>00000000</td>
<td>VR15</td>
<td>00000000</td>
</tr>
<tr>
<td>VR17</td>
<td>00000000</td>
<td>VR18</td>
<td>00000000</td>
<td>VR19</td>
<td>00000000</td>
<td>VR20</td>
<td>00000000</td>
<td>VR21</td>
<td>00000000</td>
<td>VR22</td>
<td>00000000</td>
<td>VR23</td>
<td>00000000</td>
<td>VR24</td>
<td>00000000</td>
</tr>
<tr>
<td>VR26</td>
<td>00000000</td>
<td>VR27</td>
<td>00000000</td>
<td>VR28</td>
<td>00000000</td>
<td>VR29</td>
<td>00000000</td>
<td>VR30</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Information for enclave main

Information for thread 0000000000000000

Registers on Entry to CEE3DMP:

<table>
<thead>
<tr>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
<th>VR</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR0</td>
<td>00000000</td>
<td>VR1</td>
<td>00000000</td>
<td>VR2</td>
<td>00000000</td>
<td>VR3</td>
<td>00000000</td>
<td>VR4</td>
<td>00000000</td>
<td>VR5</td>
<td>00000000</td>
<td>VR6</td>
<td>00000000</td>
<td>VR7</td>
<td>00000000</td>
</tr>
<tr>
<td>VR8</td>
<td>00000000</td>
<td>VR9</td>
<td>00000000</td>
<td>VR10</td>
<td>00000000</td>
<td>VR11</td>
<td>00000000</td>
<td>VR12</td>
<td>00000000</td>
<td>VR13</td>
<td>00000000</td>
<td>VR14</td>
<td>00000000</td>
<td>VR15</td>
<td>00000000</td>
</tr>
<tr>
<td>VR17</td>
<td>00000000</td>
<td>VR18</td>
<td>00000000</td>
<td>VR19</td>
<td>00000000</td>
<td>VR20</td>
<td>00000000</td>
<td>VR21</td>
<td>00000000</td>
<td>VR22</td>
<td>00000000</td>
<td>VR23</td>
<td>00000000</td>
<td>VR24</td>
<td>00000000</td>
</tr>
<tr>
<td>VR26</td>
<td>00000000</td>
<td>VR27</td>
<td>00000000</td>
<td>VR28</td>
<td>00000000</td>
<td>VR29</td>
<td>00000000</td>
<td>VR30</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Traceback:

<table>
<thead>
<tr>
<th>DSP</th>
<th>Entry</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>_cump</td>
<td>+000017C</td>
<td>CEEEV003</td>
<td>MODULE1</td>
<td></td>
<td>01900e</td>
<td>Call</td>
</tr>
<tr>
<td>2</td>
<td>func1</td>
<td>+000006E</td>
<td>5</td>
<td>MODULE1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>@FECBMODULE1</td>
<td>-002BD99E</td>
<td></td>
<td></td>
<td></td>
<td>01900B</td>
<td>Call</td>
</tr>
<tr>
<td>4</td>
<td>@@FECBMODULE1</td>
<td>+0000000</td>
<td>CEEEV003</td>
<td>CSAMPLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>main</td>
<td>+0000092</td>
<td>64</td>
<td>CSAMPLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>EDCZMINV</td>
<td>+0000000</td>
<td>CEEEV003</td>
<td>CSAMPLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CEEEBXRT</td>
<td>+0000018</td>
<td>CEEEV003</td>
<td>CEEEBXRT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DSA DSA Addr E Addr PU Addr PU Offset Comp Date Compile Attributes
1 20F3B6B6 20E67718 20E67718 +00000027C 20662152 LIBRARY EBCDIC HFP
2 20F3B46B 20903310 20903310 +00000066 20970314 C/C++
3 20F3B378 20FEBF90 20FEBF90 -0028099E LIBRARY
4 20F3CB20 20C0E3C0 20C0E488 +0000001A 12/15/96 LIBRARY
5 20F3C2B6 20901078 20901078 +00000099 20970314 C/C++
6 20F3B0F0 20E994E0 20E994E0 +0000002C 20662152 LIBRARY
7 20F3CB30 20992208 20992208 +00000018 20662152 CEL

Fully Qualified Names
DSA Entry Program Unit Load Module
2 funcI POSIX.CRTL.C(MODULEI) MODULEI
5 main POSIX.CRTL.C(CSAMPLE) CSAMPLE

Parameters, Registers, and Variables for Active Routines:

Control Blocks for Active Routines:

Control Blocks Associated with the Thread:

C++ C++ CAI information:

Enclave variables:

Signal information:

z/OS Language Environment Debugging Guide
Finding C/C++ information in a Language Environment dump

When a Language Environment traceback or dump is generated for a COBOL routine, information is provided that is unique to COBOL routines. COBOL-specific information includes:

- Control block information for active routines
- Condition information for active routines
- Enclave level data

Each of the unique COBOL sections of the Language Environment dump are described in Table 38.

Table 38. Contents of the COBOL sections of Language Environment

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Storage for Active Routines</td>
<td>Shows the DSAs for the active C and C++ routines. To relate a DSA frame to a particular function name, use the address associated with the frame to find the corresponding DSA. In this example, the function func1 DSA address is X'20FCB468'.</td>
</tr>
</tbody>
</table>
| [2] Control Blocks Associated with the Active Thread | Contains the following information:
  - Fields from the CAA
  - Fields specific from the CTHD and CEDB
  - Signal information |
| [2A] C/C++ CAA Fields | Contains several fields that the C/C++ programmer can use to find information about the runtime environment. For each COBOL program, there is a C-C++ Specific Thread area and a C-C++ Specific Enclave area. |
| [2B] C-C++ Specific CAA | The C-C++ specific CAA fields that are of interest to users are described below. |

**errno value**

A variable used to display error information. Its value can be set to a positive number that corresponds to an error message. The functions perror() and strerror() print the error message that corresponds to the value of errno.

**Memory file control block**

You can use the memory file control block (MFCB) to locate additional information about memory files. This control block resides at the COBOL thread level. For more information about the MFCB, see “Memory file control block” on page 201.

**Open FCB chain**

A pointer to the start of a linked list of open file control blocks (FCBs). For more information about FCBs, see File Control Block Information.
Table 38. Contents of the COBOL sections of Language Environment (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[3] Signal Information</strong></td>
<td>When the POSIX(OFF) runtime option is specified, signal information is provided in the dump to aid you in debugging. For each signal that is disabled with <code>SIG_IGN</code>, an entry value of 00000001 is made in the first field of the Signal Information field for the specified signal name.</td>
</tr>
<tr>
<td></td>
<td>For each signal that has a handler registered, the signal name and the handler name are listed. If the handler is a fetched C function, the value <code>@@FECB</code> is entered as the function name and the address of the fetched pointer is in the first field.</td>
</tr>
<tr>
<td></td>
<td>If you compile a C routine as NORENT, the WSA address is not available (N/A). For more information about the <code>signal</code> function, see z/OS XL C/C++ Programming Guide.</td>
</tr>
<tr>
<td><strong>[4] WSA Address</strong></td>
<td>The WSA Address is the base address of the writable static area which is available for all C and C++ programs except C programs compiled with the NORENT compile option.</td>
</tr>
<tr>
<td><strong>[5] atexit() Information</strong></td>
<td>Lists the functions registered with the <code>atexit()</code> function that would be run at normal termination. The functions are listed in chronological order of registration.</td>
</tr>
<tr>
<td></td>
<td>If you compile a C routine as NORENT, the WSA address is not available (N/A). For more information about the <code>atexit()</code> function, see z/OS XL C/C++ Runtime Library Reference.</td>
</tr>
<tr>
<td><strong>[6] fetch() Information</strong></td>
<td>Shows information about modules that you have dynamically loaded using <code>fetch()</code>. For each module that was fetched, the <code>fetch()</code> pointer and the function pointer are included.</td>
</tr>
<tr>
<td></td>
<td><code>ptr1 = fetch(&quot;MOD&quot;);</code></td>
</tr>
<tr>
<td></td>
<td>If you compile a C routine as NORENT, the WSA address is not available (N/A). For more information about the <code>fetch()</code> function, see z/OS XL C/C++ Programming Guide.</td>
</tr>
</tbody>
</table>
### Table 38. Contents of the COBOL sections of Language Environment (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [7] File Control Block Information | Includes the file control block (FCB) information for each C/C++ file. The FCB contains file status and attributes for files open during C/C++ active routines. You can use this information to find the data set or file name. The FCB is a handle that points to the following file information, which is displayed when applicable, for the file:  
   • Access method control block (ACB) address  
   • Data control block (DCB) address  
   • Data control block extension (DCBE) address  
   • Job file control block (JFCB) address  
   • RPL address  
   • Current buffer address  
   • Saved buffer address  
   • ddname  

   Not all FCB fields are always filled in. For example, RPLs are used only for VSAM data sets. The ddname field contains blanks if it is not used.

   The save block buffer represents auxiliary buffers that are used to save the contents of the main buffers. Such saving occurs only when a reposition is performed and there is new data; for example, an incomplete text record or an incomplete fixed-block standard (FBS) block in the buffers that cannot be flushed out of the system.

   Because the main buffers represent the current position in the file, while the save buffers merely indicate a save has occurred, check the save buffers only if data appears to be missing from the external device and is not found in the main buffers. Also, do not infer that the presence of save buffers means that data present there belongs at the end of the file. (The buffers remain, even when the data is eventually written.)

   For information about the job file control block, see z/OS MVS Data Areas in the [z/OS Internet library](www.ibm.com/systems/z/os/zos/library/bkserv).

| [8] Information for __amrc     | __amrc is a structure defined in the stdio.h header file to assist in determining errors resulting from I/O operations. The contents of __amrc can be checked for system information, such as the return code for VSAM. Certain fields of the __amrc structure can provide useful information about what occurred previously in your routine. For more information about __amrc, see "Debugging C/C++ programs" on page 173 and to z/OS XL C/C++ Programming Guide. |

| [9] Errno Information          | Shows the thread ID of the thread that generated the dump and the settings of the errno and errnojr variables for that thread. Both the errno and the errnojr variables contain the return code of the last failing z/OS UNIX system service call. These variables provide z/OS UNIX application programs access to diagnostic information returned from an underlying z/OS UNIX callable service. For more information on these return and reason codes, see z/OS UNIX System Services Messages and Codes. |

---

### Memory file control block

This section of the dump holds the following memory file control block information for each memory file the routine uses. A sample memory file control block is shown in Figure 59 on page 202.

**Memory file name**

The name assigned to this memory file.

**First memory data space**

A dump of the first 1K maximum of actual user data associated with this memory file.
Additional Floating-Point registers

The Language Environment dump formats Additional Floating Point (AFP) registers and Floating Point Control (FPC) registers when the AFP suboption of the FLOAT XL C/C++ compiler option is specified and the registers are needed. These floating-point registers are displayed in three sections of the CEEDUMP: Registers on Entry to CEE3DMP; Parameters, Registers, and Variables; and Condition Information for Active Routines. Samples of each section are given. For information on the FLOAT XL C/C++ compiler option, see z/OS XL C/C++ User’s Guide.

Registers on entry to CEE3DMP: This section of the Language Environment dump displays the sixteen floating-point registers. Figure 60 shows sample output. Note that the high half of general purpose register 14 at entry to CEE3DMP is not available and is shown in the dump as ********.

Parameters, registers, and variables for active routines: This section of the Language Environment dump displays the non-volatile floating-point registers that are saved in the stack frame. The registers are only displayed if the program owning the stack frame saved them. Dashes are displayed in the registers when the register values are not saved. A sample output is shown.
Condition information for active routines: This section of the Language Environment dump displays the floating-point registers when they are saved in the machine state; Figure 62 shows sample output.

Vector registers
The Language Environment dump formats vector registers when the vector registers are needed. These vector registers are displayed in three sections of the CEE3DMP: Registers on Entry to CEE3DMP; Parameters, Registers, and Variables; and Condition Information for Active Routines. Samples of each section are given.

Registers on entry to CEE3DMP: This section of the Language Environment dump displays the 32 vector registers. Figure 63 on page 204 shows sample output.
Figure 63. Registers on entry to CEE3DMP

Parameters, registers, and variables for active routines: This section of the Language Environment dump displays the non-volatile vector registers that are saved in the stack frame. The registers are only displayed if the program owning the stack frame saved them. Asterisks are displayed in the registers when the register values are not saved. A sample output is shown.

Parameters, Registers, and Variables for Active Routines:

Figure 64. Parameters, registers, and variables for active routines

Condition information for active routines: This section of the Language Environment dump displays the vector registers when they are saved in the
Condition Information for Active Routines

The programs tranmain and trandll were used to produce a Language Environment dump. The Language Environment dump produced by running these program is shown in “Example: dump of calling between XPLINK and non-XPLINK programs” on page 207. The dump shows XPLINK-compiled routines calling NOXPLINK-compiled routines, and NOXPLINK-compiled routines calling XPLINK-compiled routines. The program tranmain was compiled XPLINK and trandll was compiled NOXPLINK. Each was link-edited as a separate program object with the sideedek from the other.

Explanations for some of the sections are in “Finding XPLINK information in a Language Environment dump” on page 209.

Figure 65. Condition information for active routines
```c
#pragma runopts(TRACE(ON,1M,NODUMP,LE=1),XPLINK(ON),TERMTHDACT(UADUMP))
#include <stdio.h>
#pragma export(tran2)

int tran1(int, int, int, long double, int);
int tran3(int, int, long double, int);

void main(void) {
  int parm1 = 0x11111111;
  int parm2 = 0x22222222;
  int parm3 = 0x33333333;
  long double parm4 = 1234.56789;
  int parm5 = 0x55555555;
  int retval;

  printf("Main: Call Tran1\n");
  retval = tran1(parm1, parm2, parm3, parm4, parm5);
  printf("Main: Return value from Tran1 = %d\n", retval);
}

int tran2(int parm1, int parm2, int parm3, long double parm4, int parm5) {
  int retval;

  printf("Tran2: Call Tran3\n");
  retval = tran3(parm1, parm2, parm3, parm4, parm5);
  printf("Tran2: Return value from Tran3 = %d\n", retval);
  return retval;
}

int tran3(int parm1, int parm2, int parm3, long double parm4, int parm5) {
  _INT4 code, timing;

  code = 1001; /* Abend code to issue */
  timing = 1;
  printf("Tran3: About to ABEND\n");
  CEE3ABD(&code, &timing);

  return parm1 + parm2 + parm3;
}
```

**Figure 66. Sample XPLINK-compiled program (tranmain) which calls a NOXPLINK-compiled program**

```c
#include <stdio.h>
#include <ctest.h>
#include <leawi.h>
#pragma export(tran1)
#pragma export(tran3)

int tran2(int, int, int, long double, int);
int tran1(int parm1, int parm2, int parm3, long double parm4, int parm5) {
  int retval;

  printf("Tran1: Call Tran2\n");
  retval = tran2(parm1, parm2, parm3, parm4, parm5);
  printf("Tran1: Return value from Tran2 = %d\n", retval);
  return retval;
}

int tran3(int parm1, int parm2, int parm3, long double parm4, int parm5) {
  _INT4 code, timing;

  code = 1001; /* Abend code to issue */
  timing = 1;
  printf("Tran3: About to ABEND\n");
  CEE3ABD(&code, &timing);

  return parm1 + parm2 + parm3;
}
```

**Figure 67. Sample NOXPLINK-compiled program (trandll) which calls an XPLINK-compiled program**
Example dump of calling between XPLINK and non-XPLINK
programs
This article displays an example dump of calling between XPLINK and
non-XPLINK programs.
CEE3DMP V1 R12.0: Condition processing resulted in the unhandled condition.
03/18/10 3:58:00 PM
ASID: 0041 Job ID: JOB26310 Job name: XNTRAN
Step name: STEP1
UserID: HEALY

Page:

1

CEE3845I CEEDUMP Processing started.
Information for enclave main
Information for thread 8000000000000000
[1]Traceback:
DSA Entry
1
CEEHDSP
2
CEL4ABD0
3
CEEHABD
4
tran3
5
CEEVRONU
6
tran2
7
CEEVROND
8
tran1
9
CEEVRONU
10
main
11
CEEVROND
12
EDCZHINV
13
CEEBBEXT

E Offset
+00004030
+0000024C
+00000074
+000000D6
+00001026
+00000070
+000011FA
+000000F2
+00001026
+0000008C
+000011FA
+000000B4
+000001B6
E Addr
209C4238
20AFCA08
209BFCD0
212BB8C0
20ACCA58
209000E8
20ACAAA0
212BBA40
20ACCA58
20900218
20ACAAA0
20C386A8
20992208

Statement

26
27
14
18

DSA
1
2
3
4
5
6
7
8
9
10
11
12
13

DSA Addr
2110CB08
2110CA60
2110C9C8
2110C910
2110C750
212B6530
212B65B0
2110C500
2110C340
212B6680
212B6720
2110C0F0
2110C030

PU Addr
209C4238
20AFCA08
209BFCD0
212BB8C0
20ACCA58
209000E8
20ACAA48
212BBA40
20ACCA58
20900218
20ACAA48
20C386A8
20992208

Fully
DSA
4
6
8
10

Qualified Names
Entry
Program Unit
tran3
./trandll.c
tran2
./tranmain.c
tran1
./trandll.c
main
./tranmain.c

Load Mod
CEEPLPKA
CEEPLPKA
CEEPLPKA
XNTDLL
CEEPLPKA
XNTRAN
CEEPLPKA
XNTDLL
CEEPLPKA
XNTRAN
CEEPLPKA
CELHV003
CEEPLPKA
PU Offset
+00004030
+0000024C
+00000074
+000000D6
+00001026
+00000070
+00001252
+000000F2
+00001026
+0000008C
+00001252
+000000B4
+000001B6

Program Unit
CEEHDSP
CEL4ABD0
CEEHABD
trandll.c
CEEVRONU
tranmain.c

Service
D1908
D1908
D1908
D1908

trandll.c
CEEVRONU
tranmain.c

D1908

EDCZHINV
CEEBBEXT

D1908
D1908

Status
Call
Exception
Call
Call
Call
Call
Call
Call
Call
Call
Call
Call
Call

Comp Date Compile Attributes
20061215 CEL
20061215 CEL
20061215 CEL
20000505 C/C++
20061215 CEL
20000421 C/C++
XPLINK EBCDIC HFP
20061215 CEL
XPLINK EBCDIC HFP
20000505 C/C++
20061215 CEL
20000421 C/C++
XPLINK EBCDIC HFP
20061215 CEL
XPLINK EBCDIC HFP
20061214 LIBRARY
20061215 CEL
Load Module
XNTDLL
XNTRAN
XNTDLL
XNTRAN

Condition Information for Active Routines
Condition Information for CEL4ABD0 (DSA address 2110CA60)
CIB Address: 2110D428
Current Condition:
CEE0198S The termination of a thread was signaled due to an unhandled condition.
Original Condition:
CEE3250C The system or user abend U1001 R=00000000 was issued.
Location:
Program Unit: CEL4ABD0 Entry: CEL4ABD0 Statement: Offset: +0000024C
Machine State:
ILC..... 0002
Interruption Code..... 000D
PSW..... 078D1400 A0AFCC54
GPR0..... 00000000_84000000 GPR1..... 00000000_840003E9 GPR2..... 00000000_00000000 GPR3..... 00000000_2110C9B0
GPR4..... 00000000_00000001 GPR5..... 00000000_20914E10 GPR6..... 00000000_00000002 GPR7..... 00000000_00000000
GPR8..... 00000000_A0900003 GPR9..... 00000000_212BB868 GPR10.... 00000000_209BFDAC GPR11.... 00000000_20AFCA08
GPR12.... 00000000_209139B0 GPR13.... 00000000_2110CA60 GPR14.... 00000000_A09BFD46 GPR15.... 00000000_00000000
ABEND code: 000003E9 Reason code: 00000000
Storage dump near condition, beginning at location: 20AFCC44
+000000 20AFCC44 88100008 41000084 89000018 16100A0D 58D0D004 98ECD00C 07FE0000 D7C1E3C3 |h......di...........q.......PATC|
GPREG STORAGE:
Storage around GPR0 (04000000)
-0020 03FFFFE0
Inaccessible storage.
+0000 04000000
Inaccessible storage.
+0020 04000020
Inaccessible storage.
.
.
.
[2]Parameters, Registers, and Variables for Active Routines:
tran3 (DSA address 2110C910):
UPSTACK DSA
Parameters:
parm5
signed int
1431655765
parm4
long double
1.234567889999999977135303197E+03
parm3
signed int
858993459
parm2
signed int
572662306
parm1
signed int
286331153
Saved Registers:
GPR0..... 20914D70 GPR1..... 2110C9A8 GPR2..... 2110C9B4 GPR3..... 212BB8FA
GPR4..... 2110C9B0 GPR5..... 20914E10 GPR6..... 00000000 GPR7..... 00000000
GPR8..... A0900003 GPR9..... 212BB868 GPR10.... 212BB8C0 GPR11.... 20ACDF1C
GPR12.... 209139B0 GPR13.... 2110C910 GPR14.... A12BB998 GPR15.... A09BFCD0
GPREG STORAGE:
Storage around GPR0 (20914D70)
-0020 20914D50 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |................................|
+0000 20914D70 180F58FF 001007FF 212BB3A0 20914D70 212BB3B0 00000000 00000000 00000000 |.............j(.................|
+0020 20914D90 180F58FF 001007FF 212BB0B0 20914D70 212BB3B0 00000000 00000000 00000000 |.............j(.................|

Chapter 4. Debugging C/C++ routines

207


[Control Blocks for Active Routines]

DSA for task 3: 210C1C0
- 000000 FLAGS... 00000000 R12...... R13...... R14...... R15......
- 000001 R12...... R13...... R14...... R15......
- 000002 R12...... R13...... R14...... R15......
- 000003 R12...... R13...... R14...... R15......
- 000004 NAB... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13...
- 000005 reserved. 209000B8 reserved. 209000B8 reserved. 209000B8 reserved.
- 000006 reserved. 20914E70 reserved. 20914E70 reserved. 20914E70 reserved.
- 000007 reserved. 2091BD6B reserved. 2091BD6B reserved. 2091BD6B reserved.

DSA for CEERVND: 210C750
- 000000 FLAGS... 00000000 R12...... R13...... R14...... R15......
- 000001 R12...... R13...... R14...... R15......
- 000002 R12...... R13...... R14...... R15......
- 000003 R12...... R13...... R14...... R15......
- 000004 NAB... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13...
- 000005 reserved. 209000B8 reserved. 209000B8 reserved. 209000B8 reserved.
- 000006 reserved. 20914E70 reserved. 20914E70 reserved. 20914E70 reserved.
- 000007 reserved. 2091BD6B reserved. 2091BD6B reserved. 2091BD6B reserved.

DSA for task 3: 210C6B0
- 000000 HA... 210C6B0 R12...... R13...... R14...... R15......
- 000001 R12...... R13...... R14...... R15......
- 000002 R12...... R13...... R14...... R15......
- 000003 R12...... R13...... R14...... R15......
- 000004 NAB... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13...
- 000005 reserved. 209000B8 reserved. 209000B8 reserved. 209000B8 reserved.
- 000006 reserved. 20914E70 reserved. 20914E70 reserved. 20914E70 reserved.
- 000007 reserved. 2091BD6B reserved. 2091BD6B reserved. 2091BD6B reserved.

DSA for CEERVND: 210C750
- 000000 FLAGS... 00000000 R12...... R13...... R14...... R15......
- 000001 R12...... R13...... R14...... R15......
- 000002 R12...... R13...... R14...... R15......
- 000003 R12...... R13...... R14...... R15......
- 000004 NAB... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13...
- 000005 reserved. 209000B8 reserved. 209000B8 reserved. 209000B8 reserved.
- 000006 reserved. 20914E70 reserved. 20914E70 reserved. 20914E70 reserved.
- 000007 reserved. 2091BD6B reserved. 2091BD6B reserved. 2091BD6B reserved.

DSA for task 3: 210C6B0
- 000000 HA... 210C6B0 R12...... R13...... R14...... R15......
- 000001 R12...... R13...... R14...... R15......
- 000002 R12...... R13...... R14...... R15......
- 000003 R12...... R13...... R14...... R15......
- 000004 NAB... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13... 210C9C8 R13...
- 000005 reserved. 209000B8 reserved. 209000B8 reserved. 209000B8 reserved.
- 000006 reserved. 20914E70 reserved. 20914E70 reserved. 20914E70 reserved.
- 000007 reserved. 2091BD6B reserved. 2091BD6B reserved. 2091BD6B reserved.
Finding XPLINK information in a Language Environment dump

Table 39 describes the specific XPLINK information in sections of the Language Environment dump.

Table 39. Contents of XPLINK information in a Language Environment dump

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Traceback</td>
<td>When an XPLINK-compiled routine calls a NOXPLINK-compiled routine, a glue routine gets control to convert the linkage conventions of the XPLINK caller to those of the NOXPLINK callee. In the sample dump, this routine is CEEVRONU and it appears between main() and tran1() and again between tran2() and tran3(). When a NOXPLINK-compiled routine calls an XPLINK-compiled routine, a glue routine gets control to convert the linkage conventions of the NOXPLINK caller to those of the XPLINK callee. In the sample dump, this routine is CEEVROND and it appears between EDCZHINV and main() and again between tran1() and tran2().</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[2] Parameters, Registers, and Variables for Active Routines</th>
<th>In this section, each DSA is identified as one of the following:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>UPSTACK DSA</strong></td>
</tr>
<tr>
<td></td>
<td>The DSA format is that for a NOXPLINK-compiled program that uses an upward growing stack.</td>
</tr>
<tr>
<td></td>
<td><strong>DOWNSTACK DSA</strong></td>
</tr>
<tr>
<td></td>
<td>The DSA format is that for a XPLINK-compiled program that uses a downward growing stack.</td>
</tr>
<tr>
<td></td>
<td><strong>TRANSITION DSA</strong></td>
</tr>
<tr>
<td></td>
<td>The DSA format is that of its callee. A transition DSA can occur between an UPSTACK DSA and a DOWNSTACK DSA where it represents a transition from one linkage convention to another. A transition DSA can also occur between two DOWNSTACK DSAs where it represents a transition from one stack segment to another (a stack overflow).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[3] Control Blocks for Active Routines</th>
<th>In this section, DSAs are formatted. Those previously identified as UPSTACK DSAs will have one format and those identified as DOWNSTACK DSAs will have a different format. Those identified as TRANSITION DSAs will have two parts; the first will be either the downstack or upstack format, the second is unique to transition DSAs and contains information about the transition.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>It is important to understand that the registers saved in an upstack DSA are those saved by a routine that the DSA-owning routine called. Typically register 15 is the entry point of the routine that was called, and register 14 is the return address into the DSA-owning routine. In contrast, the registers saved in a downstack DSA are those saved by the DSA-owning routine on entry. Register 7 is the return address back to the caller of the DSA-owning routine. Register 6 may be the entry point of the DSA-owning routine. (This is not true when the Branch Relative and Save instruction is used to implement the call.)</td>
</tr>
</tbody>
</table>

C/C++ contents of the Language Environment trace tables

Language Environment provides the following C/C++ trace table entry types that contain character data. For more information about the Language Environment trace table format, see “Understanding the trace table entry (TTE)” on page 161.

- Trace entry 1 occurs when a base C library function is called.
• Trace entry 2 occurs when a base C library function returns.
• Trace entry 3 occurs when a POSIX C library function is called.
• Trace entry 4 occurs when a POSIX C library function returns.
• Trace entry 5 occurs when an XPLINK base C or POSIX C library function is called.
• Trace entry 6 occurs when an XPLINK base C or POSIX C library function returns.
• Trace entry 7 occurs when an XPLINK function calls a non-XPLINK function.
• Trace entry 8 occurs when a non-XPLINK function calls an XPLINK function.

The format for trace table entry 1 is:

```
NameOfCallingFunction
  --> (xxx) NameOfCalledFunction
```

or, for called functions calloc, free, malloc, and realloc:

```
NameOfCallingFunction
  --> (xxx) NameOfCalledFunction<(input_parameters)>
```

In addition, when the call is due to one of these C++ operators:
- new,
- new[],
- delete,
- delete[]

then, the C++ operator will appear and the format becomes:

```
NameOfCallingFunction
  --> (xxx) NameOfCalledFunction<(input_parameters)>
NameOfC++Operator
```

The format for trace table entry 2 is:

```
<-- (xxx) R15=value ERRNO=value
```

The format for trace table entry 3 is:

```
NameOfCallingFunction
  --> (xxx) NameOfCalledFunction
```

The format for trace table entry 4 is:

```
<-- (xxx) R15=value ERRNO=value ERRNO2=value
```

The format for trace table entry 5, which is shown below, is just like trace table entry 1. The input_parameters and NameOfC++Operator only appear for the appropriate functions. The angle brackets (<>​) indicate that this information does
not always appear.

The format for trace table entry 6 is:

```
<--(xxx) R1=xxxxxxxx R2=xxxxxxxx R3=xxxxxxxx ERRNO=xxxxxxxx ERRNO2=xxxxxxxx
```

In all entry types, (xxx) and (xxxx) are numbers associated with the called library function and are used to associate a specific entry record with its corresponding return record.

For entry types 5 and 6, the number will be the same as the number of the function as seen in the C runtime library definition side-deck, SCEELIB dataset member CELHS003, on the IMPORT statement for that function.

The format for trace table entry 7 is:

```
ModuleNameOfCallingFunction:NameOfCallingXplinkFunction
---ModuleNameOfCalledFunction:NameOfCalledNonXplinkFunction
```

The format for trace table entry 8 is:

```
ModuleNameOfCallingFunction:NameOfCallingNonXplinkFunction
---ModuleNameOfCalledFunction:NameOfCalledXplinkFunction
```

For entry types 7 and 8, 16 bytes is for the module name and 32 bytes is for the function name. If the name is longer than 16 or 32 bytes, an extra trace entry is taken. The name is truncated and only the first 32/64 (16/32) bytes will appear in the trace table entry. Also, a module name might not always be located, such as when a DLL is freed. If that occurs, "UNKNOWN" appears for the module name in the trace table entry.

The below trace table shows a non-XPLINK trace that has examples of C/C++ trace table entry types 1 thru 4.

<table>
<thead>
<tr>
<th>Language Environment Trace Table:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most recent trace entry is at displacement: 020500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Displacement</th>
<th>Trace Entry in Hexadecimal</th>
<th>Trace Entry in EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>Time 20.52.46.666280</td>
<td>main</td>
</tr>
<tr>
<td></td>
<td>Date 2001.08.26</td>
<td>--&gt;(139) strcpy()</td>
</tr>
<tr>
<td>+000100</td>
<td>Time 20.52.46.666289</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Date 2001.08.26</td>
<td></td>
</tr>
<tr>
<td>+000110</td>
<td>Time 20.52.46.666289</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 4. Debugging C/C++ routines 211
The code below shows an XPLINK trace that has examples of the trace entries 5 and 6.

Language Environment Trace Table:

Most recent trace entry is at displacement: 000008

<table>
<thead>
<tr>
<th>Displacement</th>
<th>Trace Entry in Hexadecimal</th>
<th>Trace Entry in EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>Time 22.41.35.433944 Date 2001.08.30 Thread ID... 26C70000000000</td>
<td></td>
</tr>
<tr>
<td>+000010</td>
<td>Member ID... 03 Flags... 0000000 Entry Type... 00000006</td>
<td></td>
</tr>
<tr>
<td>+000080</td>
<td>Time 22.41.35.433948 Date 2001.08.30 Thread ID... 26C70000000000</td>
<td></td>
</tr>
<tr>
<td>+000090</td>
<td>Member ID... 03 Flags... 0000000 Entry Type... 00000005</td>
<td></td>
</tr>
<tr>
<td>+000098</td>
<td>C9D9E3D3 D98A95D6 7A7A1C9 09E3D90 5B2A96A 9B3B54A0 5B040404</td>
<td>IRTLResource::IRTLResource()</td>
</tr>
<tr>
<td>+0000A0</td>
<td>00400040 04040040 40400440 40400440 40400440 40400440</td>
<td></td>
</tr>
<tr>
<td>+000098</td>
<td>C9D9E3D3 D98A95D6 7A7A1C9 09E3D90 5B2A96A 9B3B54A0 5B040404</td>
<td>IRTLResource::IRTLResource()</td>
</tr>
<tr>
<td>+0000A0</td>
<td>00400040 04040040 40400440 40400440 40400440 40400440</td>
<td></td>
</tr>
<tr>
<td>+0000A8</td>
<td>Time 22.41.35.433957 Date 2001.08.30 Thread ID... 26C70000000000</td>
<td></td>
</tr>
<tr>
<td>+000100</td>
<td>Member ID... 03 Flags... 0000000 Entry Type... 00000006</td>
<td></td>
</tr>
<tr>
<td>+000110</td>
<td>Time 22.41.35.433959 Date 2001.08.30 Thread ID... 26C70000000000</td>
<td></td>
</tr>
<tr>
<td>+000120</td>
<td>Member ID... 03 Flags... 0000000 Entry Type... 00000006</td>
<td></td>
</tr>
<tr>
<td>+000128</td>
<td>00400040 04040040 40400440 40400440 40400440 40400440</td>
<td></td>
</tr>
<tr>
<td>+000138</td>
<td>Time 22.41.35.433963 Date 2001.08.30 Thread ID... 26C70000000000</td>
<td></td>
</tr>
<tr>
<td>+000200</td>
<td>Member ID... 03 Flags... 0000000 Entry Type... 00000005</td>
<td></td>
</tr>
<tr>
<td>+000210</td>
<td>Time 22.41.35.433967 Date 2001.08.30 Thread ID... 26C70000000000</td>
<td></td>
</tr>
<tr>
<td>+000300</td>
<td>Member ID... 03 Flags... 0000000 Entry Type... 00000006</td>
<td></td>
</tr>
<tr>
<td>+000310</td>
<td>Time 22.41.35.433972 Date 2001.08.30 Thread ID... 26C70000000000</td>
<td></td>
</tr>
<tr>
<td>+000380</td>
<td>Member ID... 03 Flags... 0000000 Entry Type... 00000005</td>
<td></td>
</tr>
<tr>
<td>+000390</td>
<td>Time 22.41.35.433974 Date 2001.08.30 Thread ID... 26C70000000000</td>
<td></td>
</tr>
</tbody>
</table>

Figure 68 on page 214 shows an example of the format of the trace table entry type 7 and 8.
Figure 68. Trace table with trace table entry types 7 and 8

The following is an example of a dump of the trace table when you specify the LE=20 suboption.

Language Environment Trace Table:

Most recent trace entry is at displacement: 000080

<table>
<thead>
<tr>
<th>Displacement</th>
<th>Trace Entry in Hexadecimal</th>
<th>Trace Entry in EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>+00000</td>
<td>21405.06.799195 2005.05.01 Thread ID... 21D8C30000000000</td>
<td>CELHV003 :ECDZHNV --&gt;a859c41x</td>
</tr>
<tr>
<td>+00100</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000000</td>
<td>CEEPLPKA :CEEPLDCL --&gt;CEELV03</td>
</tr>
<tr>
<td>+00200</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000007</td>
<td></td>
</tr>
<tr>
<td>+00300</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000008</td>
<td></td>
</tr>
<tr>
<td>+00400</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000009</td>
<td></td>
</tr>
<tr>
<td>+00500</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000000A</td>
<td></td>
</tr>
<tr>
<td>+00600</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000000B</td>
<td></td>
</tr>
<tr>
<td>+00700</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000000C</td>
<td></td>
</tr>
<tr>
<td>+00800</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000000D</td>
<td></td>
</tr>
<tr>
<td>+00900</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000000E</td>
<td></td>
</tr>
<tr>
<td>+01000</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000000F</td>
<td></td>
</tr>
<tr>
<td>+01100</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000010</td>
<td></td>
</tr>
<tr>
<td>+01200</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000011</td>
<td></td>
</tr>
<tr>
<td>+01300</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000012</td>
<td></td>
</tr>
<tr>
<td>+01400</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000013</td>
<td></td>
</tr>
<tr>
<td>+01500</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000014</td>
<td></td>
</tr>
<tr>
<td>+01600</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000015</td>
<td></td>
</tr>
<tr>
<td>+01700</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000016</td>
<td></td>
</tr>
<tr>
<td>+01800</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000017</td>
<td></td>
</tr>
<tr>
<td>+01900</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000018</td>
<td></td>
</tr>
<tr>
<td>+02000</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000019</td>
<td></td>
</tr>
<tr>
<td>+02100</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000001A</td>
<td></td>
</tr>
<tr>
<td>+02200</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000001B</td>
<td></td>
</tr>
<tr>
<td>+02300</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000001C</td>
<td></td>
</tr>
<tr>
<td>+02400</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000001D</td>
<td></td>
</tr>
<tr>
<td>+02500</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000001E</td>
<td></td>
</tr>
<tr>
<td>+02600</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000001F</td>
<td></td>
</tr>
<tr>
<td>+02700</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000020</td>
<td></td>
</tr>
<tr>
<td>+02800</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000021</td>
<td></td>
</tr>
<tr>
<td>+02900</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000022</td>
<td></td>
</tr>
<tr>
<td>+03000</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000023</td>
<td></td>
</tr>
<tr>
<td>+03100</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000024</td>
<td></td>
</tr>
<tr>
<td>+03200</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000025</td>
<td></td>
</tr>
<tr>
<td>+03300</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000026</td>
<td></td>
</tr>
<tr>
<td>+03400</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000027</td>
<td></td>
</tr>
<tr>
<td>+03500</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000028</td>
<td></td>
</tr>
<tr>
<td>+03600</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000029</td>
<td></td>
</tr>
<tr>
<td>+03700</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000002A</td>
<td></td>
</tr>
<tr>
<td>+03800</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000002B</td>
<td></td>
</tr>
<tr>
<td>+03900</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000002C</td>
<td></td>
</tr>
<tr>
<td>+04000</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000002D</td>
<td></td>
</tr>
<tr>
<td>+04100</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000002E</td>
<td></td>
</tr>
<tr>
<td>+04200</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000002F</td>
<td></td>
</tr>
<tr>
<td>+04300</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000030</td>
<td></td>
</tr>
<tr>
<td>+04400</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000031</td>
<td></td>
</tr>
<tr>
<td>+04500</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000032</td>
<td></td>
</tr>
<tr>
<td>+04600</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000033</td>
<td></td>
</tr>
<tr>
<td>+04700</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000034</td>
<td></td>
</tr>
<tr>
<td>+04800</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000035</td>
<td></td>
</tr>
<tr>
<td>+04900</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000036</td>
<td></td>
</tr>
<tr>
<td>+05000</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000037</td>
<td></td>
</tr>
<tr>
<td>+05100</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000038</td>
<td></td>
</tr>
<tr>
<td>+05200</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 00000039</td>
<td></td>
</tr>
<tr>
<td>+05300</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000003A</td>
<td></td>
</tr>
<tr>
<td>+05400</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000003B</td>
<td></td>
</tr>
<tr>
<td>+05500</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000003C</td>
<td></td>
</tr>
<tr>
<td>+05600</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000003D</td>
<td></td>
</tr>
<tr>
<td>+05700</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000003E</td>
<td></td>
</tr>
<tr>
<td>+05800</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000003F</td>
<td></td>
</tr>
</tbody>
</table>

214 z/OS Language Environment Debugging Guide
Debugging examples of C/C++ routines

This section contains examples that demonstrate the debugging process for C/C++ routines. Important areas of the output are highlighted. Data unnecessary to the debugging examples has been replaced by ellipses.

**Divide-by-zero error**

![Figure 69 on page 216](image)

The code was compiled with RENT so static and external variables need to be calculated from the WSA field. The code was compiled with XREF, LIST and OFFSET to generate a listing, which is used to calculate addresses of functions and data. The code was processed by the binder with MAP to generate a binder map, which is used to calculate the addresses of static and external variables.
To debug this routine, use the following steps:

1. Locate the Original Condition message in the Condition Information for Active Routines section of the dump. In this example, the message is CEE3209S. The system detected a fixed—point divide exception. This message indicates the error was caused by an attempt to divide by zero. For more information about CEE3209S, see z/OS Language Environment Runtime Messages.

The traceback section of the dump indicates that the exception occurred at offset X'76' within function funcb. This information is used along with the compiler-generated Pseudo Assembly Listing to determine where the problem occurred.

If the TEST compiler option is specified, variable information is in the dump. If the GONUMBER compiler option is specified, statement number information is in the dump. Figure 70 on page 217 shows the generated traceback from the dump.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int statint = 73;
int fa;
void funcb(int *pp);

int main(void) {
    int aa, bb=1;
    aa = bb;
    funcb(&aa);
    return(99);
}

void funcb(int *pp) {
    int result;
    fa = *pp;
    result = fa/(statint-73);
    return;
}
```

Figure 69. C routine with a divide-by-zero error
2. Locate the instruction with the divide-by-zero error in the Pseudo Assembly Listing in Figure 71 on page 218.

The offset (within func) of the exception from the traceback (X’76’) reveals the divide instruction: DR r4,r1 at that location. Instructions X’66’ through X’76’ refer to the result = fa/(statint-73); line of the C/C++ routine.
OFFSET OBJECT CODE   LINE#   FILE#  PSEUDO ASSEMBLY LISTING
000000 000005 * int funcb(int *pp) {
           000015   funcb  DS  00
.
.
000046 50D0 0D04 000015   ST  r13,4,(r14)
00004E 1D8E 000015   LR  r13,r14
00004C End of Prolong

00004C 58E0 C1F4 000000   L  r14,.CEECAAL,(r12,500)
000016 * int result;
000017   * fa = *pp;
000050 5820 1000 000017   L  r2,pp,(r1,0)
000054 5810 3062 000018   L  r1,-(statint),(r3,98)
000058 58F0 3066 000017   L  r15,-0(fa),(r3,102)
00005C C000 0000 0026 000000 LARL  r0,F38'
000062 58A0 2000 000017   L  r6,(r4,int),(r2,0)
000018 * result = fa((statint-73);
000066 5811 0D00 000018   L  r1,statint(r1,r14,0)
00006A 504F 0D00 000017   ST  r4,fa(r15,r14,0)
00006E A71A FFB7 000018   AMI  r1,H-73'
000072 BE40 0020 000018   SREZ  r4,32
000076 1D41 0000 0018   DR  r4,r1
000019 * printf("Result = %d\n",result);
000078 58F0 306A 000019   L  r15,\V\(\printf\,(r3,106)
00007C 4110 0098 000019   LA  r1,\MX\_TEMP2\(,(r13,152)
000080 5000 0098 000019   ST  r0,\MX\_TEMP2\(,(r13,152)
000084 5050 009C 000019   ST  r5,\MX\_TEMP2\(,(r13,156)
000088 00E0 0000 0019   BASR  r14,r15
000020 * return result;
00008A 18F5 0000 0020   LR  r15,r5
000021   000021   02L3  DS  0H
00008C Start of Epilog
00008C 58D0 0004 000021   L  r13,4,(r13)
000090 58E0 000C 000021   L  r14,12,(r13)
000094 9825 001C 000021   LM  r15,r5,28(r13)
000098 051E 0000 0021   BALR  r1,r14
00009A 0707 0000 0021   NDPR  7
00009C Start of Literals
00009C 00000000 000021   =Q(statint)
0000A0 00000000 000021   =Q(fa)
0000A4 00000000 000021   =V(printf)
0000A8 End of Literals
*** General purpose registers used: 00012F
*** Floating point registers used: 000170
*** Size of register spill area: 000010
*** Size of dynamic storage: 098
*** Size of executable code: 000015

Constant Area

000000 D985A2A4 93A3407E 406C8415 00 |Result = %d.. |

PPA1: Entry Point Constants
000000 ICCEA106  +F'4833036866'  Flags
000004 00000228  +A(PPA2-main)
000008 00000178  +A(PPA3-main)
00000C 00000000  +F'0'  No EPD
000010 FE000000  +F'=-33554432'  Register save mask
000014 00000000  +F'0'  Member flags
000018 90  +AL1(144)  Flags
000019 00000000  +AL3(0)  Callee's DSA use/8
00001C 0040  +H'64'  Flags
00001E 0012  +H'18'  Offset/2 to CDL
000020 00000000  +F'0'  Reserved
000024 5000003C  +F'1342177340'  CDL function length/2
000028 80FE00B8  +F'=-312'  CDL function EP offset
00002C 38260000  +F'962014464'  CDL prolog
000030 40090033  +F'1074331699'  CDL epilog
000034 00000000  +F'0'  CDL end
000038 0004 ****  AL2(4),C'main'

PPA1 End

Figure 71. Pseudo assembly listing (C/C++ routine divide-by-zero error)

218 z/OS Language Environment Debugging Guide
3. Verify the value of the divisor \texttt{statint}. The procedure specified below is to be used for determining the value of static variables only. If the divisor is an automatic variable, there is a different procedure for finding the value of the variable. For more information about finding automatic variables in a dump, see “Steps for finding automatic variables” on page 183. Because this routine was compiled with the RENT option, find the WSA address in the Enclave Control Blocks section of the dump. In this example, this address is \texttt{X'20914F50'}. Figure 72 shows the WSA address.

\begin{verbatim}
Enclave Control Blocks:
  WSA address.................20914F50
\end{verbatim}

\textbf{Figure 72. C/C++ CAA information in dump (C/C++ routine divide-by-zero error)}

4. Routines compiled with the RENT option must also be processed by the binder. The binder produces the Writable Static Map. Find the offset of \texttt{statint} in the Writable Static Map in Figure 73. In this example, the offset is \texttt{X'0'}. 

\begin{verbatim}
----------------------------------
CLASS C_WSA                     LENGTH =   AC ATTRIBUTES = MRG, DEFER, RMODE=ANY
OFFSET = 0 IN SEGMENT 002       ALIGN = DBLWORD
----------------------------------

CLASS    NAME       TYPE LENGTH SECTION
0        statint    PART  4 statint
8        fa         PART  4 fa
10       environ    PART  4 environ
18       errno      PART  4 errno
----------------------------------

Figure 73. Writable static map (C/C++ routine divide-by-zero error)
\end{verbatim}

5. Add the WSA address of \texttt{X'20914F50'} to the offset of \texttt{statint}. The result is \texttt{X'20914F50'}. This is the address of the variable \texttt{statint}, which is in the writable static area.

The writable static area is shown in the Enclave Storage section of the dump. For a load module, the writable static area is storage allocated by the C/C++ runtime for the C/C++ user, so it is in the user heap. For a program object, the writable static area is storage allocated by the loader and is shown in the WSA for Program Object(s) section of the dump.

For this example, the program was built as a program object. The writable static area is displayed in the Enclave Storage section of the dump, shown in Figure 74 on page 220.

6. To find the variable \texttt{statint} in the writable static area, locate the closest address listed that is before the address of \texttt{statint}. In this case, that address is \texttt{X'20914F50'}. Count across \texttt{X'00'} to location \texttt{X'20914F50'}. The value at that location is \texttt{X'49'} (that is, \texttt{statint} is 73), and hence the fixed point divide exception.
Calling a nonexistent non-XPLINK function

Figure 75 demonstrates the error of calling a nonexistent function. This routine was compiled with the compiler options LIST, OFFSET, and RENT and was run with the option TERMTHDACT(DUMP). The code was processed by the binder with MAP to generate a binder map, which is used to calculate the addresses of static and external variables. This routine was not compiled with the TEST(ALL) compiler option. As a result, arguments and variables do not appear in the dump.

To debug this routine, use the following steps:

1. Locate the Original Condition message in the Condition Information for Active Routines section of the dump, shown in Figure 76 on page 221. In this example, the message is CEE3201S The system detected an operation exception (System Completion Code=0C1). This message suggests that the error was caused by an attempt to branch to an unknown address. For additional information about CEE3201S, see z/OS Language Environment Runtime Messages.

The Location section of the dump indicates that the exception occurred at offset X'20900978' within function funca and that there may have been a bad branch from offset X'+0000005A' within function funca. The negative offset indicates that the offset cannot be used to locate the instruction that caused the error. Another indication of bad data is the value of X'80000002' in the instruction address of the PSW. This address indicates that an instruction in the routine branched outside the bounds of the routine.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>

void funca(int* aa);
int (*func_ptr)(void)=0;

int main(void)
{
    int aa;
    funca(&aa);
    printf("result of funca = %d\n",aa);
    return;
}

void funca(int* aa)
{
    *aa = func_ptr();
    return;
}
```

Figure 75. C/C++ example of calling a nonexistent subroutine
Figure 76. Sections of the dump from example C routine (calling a nonexistent subroutine)

2. Find the branch instructions at offset X'0000005A' of funca in the listing in [Figure 77 on page 222]. The instruction is BASR r14, r15. This branch is part of the source statement *aa = func_ptr().
3. Find the offset of `func_ptr` in the Writable Static Map, shown in Figure 78, as produced by the binder.

```
Figure 77. Pseudo assembly listing (calling a nonexistent subroutine)
```

```
Figure 78. Writable static map (calling a nonexistent subroutine)
```

4. Add the offset of `FUNC@PTR (X'0')` to the address of WSA (X'20914F58'). The result (X'20914F58') is the address of the function pointer `func_ptr` in the writable static storage area within the heap. This value is 0, indicating the variable is uninitialized. Figure 79 on page 223 shows the sections of the dump.
Calling a nonexistent XPLINK function

Figure 80 demonstrates the error of calling a nonexistent function. This routine was compiled with the compiler options XPLINK, LIST and RENT and was run with the option TERMTHDACT(DUMP). This routine was not compiled with the TEST(ALL) compile option. As a result, arguments and variables do not appear in the dump.

To debug this routine, use the following steps:

1. Locate the Original Condition message in the Condition Information for Active Routines section of the dump, shown in “Sections of the dump from example C routine (calling a nonexistent XPLINK function)” on page 225. In this example, the message is CEE3201S The system detected an operation exception (System Completion Code=0C1). This message suggests that the error was caused by an attempt to branch to an unknown address. For additional information about CEE3201S, see z/OS Language Environment Runtime Messages.

The location section of the dump indicates that the exception occurred at offset X’-20900158’ within function funca and that there may have been a bad branch from offset X’+0000001C’. The negative offset indicates that the offset cannot be used to locate the instruction that caused the error. Another indication of bad data is the value of X’80000004’ in the instruction address of the PSW. This address indicates that an instruction in the routine branched outside the bounds of the routine.

2. Find the branch instruction at offset X’+0000001C’ of funca in the listing in Figure 81 on page 224. This instruction is BASR r7, r6. This branch is part of the...
source statement *aa = func_ptr().

Figure 81. Pseudo assembly listing (calling a nonexistent XPLINK function)

3. Find the offset of func_ptr in the Writable Static Map, shown in Figure 82.

Figure 82. Writable static map (calling a nonexistent XPLINK function)

4. Add the offset of func_ptr (X'38') to the address of WSA (X'20914FC0'). The result (X'20914FF8') is the address of the function pointer func_ptr in the writable static storage area within the heap. This value is 0, indicating the variable is uninitialized. Figure 83 on page 225 shows the sections of the dump.
Enclave Control Blocks:

<table>
<thead>
<tr>
<th>DLL Information:</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSA Addr: Module Addr: Thread ID: Use Count: Name</td>
</tr>
<tr>
<td>20914FC0: 00000001: min</td>
</tr>
</tbody>
</table>

WSA address: 20914FC0

Enclave Storage:

<table>
<thead>
<tr>
<th>WSA for Program Object(s):</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSA: 20914FC0</td>
</tr>
<tr>
<td>+000080 20914FC0 C36E62E C1404400 40040400 40040400 99852A94 936A4096 8640864A 95838140</td>
</tr>
<tr>
<td>+000082 20914FE0 7F6E8C48 15000000 20914F78 00000000 00000000 20F23280 00000000 00000000</td>
</tr>
</tbody>
</table>

Figure 83. Enclave control blocks and storage sections in dump (calling a nonexistent XPLINK function)

Sections of the dump from example C routine (calling a nonexistent XPLINK function)

| AsusMP vs R12:0: Condition processing resulted in the unhandled condition. | Page: 1 |
|----------------------------------------------------------------------------|
| ASID: 0051               Job ID: 20006606     Job name: XEIXST             |
| Step name: STEPI       UserID: HEALY                           |

CEEBHASI CEDUMP Processing started.

Information for enclave main

Information for thread 0000000000000000

Traceback:

<table>
<thead>
<tr>
<th>DSA</th>
<th>E Addr</th>
<th>Program Unit</th>
<th>Macro Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EEEDSF</td>
<td>CEEPLPKA</td>
<td>CEEEDSF</td>
</tr>
<tr>
<td>2</td>
<td>CEEHMAH</td>
<td>CEEEMHIN</td>
<td>CEEHMAH</td>
</tr>
<tr>
<td>3</td>
<td>funca</td>
<td>XEIXST</td>
<td>XEIXST</td>
</tr>
<tr>
<td>4</td>
<td>main</td>
<td>XEIXST</td>
<td>XEIXST</td>
</tr>
<tr>
<td>5</td>
<td>CEEHMAH</td>
<td>CEEEMHIN</td>
<td>CEEHMAH</td>
</tr>
<tr>
<td>6</td>
<td>EDCN2HV</td>
<td>CELVH003</td>
<td>EDCN2HV</td>
</tr>
<tr>
<td>7</td>
<td>CEEBEXT</td>
<td>CEEPLPKA</td>
<td>CEEBEXT</td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Condition Information for (DSA address 21856280)

<table>
<thead>
<tr>
<th>CIW Address: 2101C2E0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Condition: CEEHMASI. The termination of a thread was signaled due to an unhandled condition.</td>
</tr>
<tr>
<td>Original Condition: CEE3201S. The system detected an operation exception (System Completion Code=0C1).</td>
</tr>
<tr>
<td>Location:</td>
</tr>
<tr>
<td>Program Unit: Entry: funca Statement: Offset: -20900158</td>
</tr>
<tr>
<td>Possible Bad Branch: Statement: Offset: +0000001C</td>
</tr>
</tbody>
</table>

Machine State:

<table>
<thead>
<tr>
<th>ILC....</th>
<th>0002</th>
<th>Intercept Code...</th>
<th>0081</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSW....</td>
<td>097D240E</td>
<td>80000002</td>
<td></td>
</tr>
<tr>
<td>GPR0....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR2....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR3....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR4....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR5....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR6....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR7....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR8....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR9....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR10....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR11....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR12....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR13....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
</tbody>
</table>

Parameter, Registers, and Variables for Active Routines:

CEEDSF (DSA address 2110C500):

UPSTACK DSA

Saved Registers:

<table>
<thead>
<tr>
<th>GPR0....</th>
<th>2110C500</th>
<th>GPR1....</th>
<th>2110C500</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR2....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR3....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR4....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR5....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR6....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR7....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR8....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR9....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR10....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR11....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR12....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR13....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR14....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
<tr>
<td>GPR15....</td>
<td>00000000</td>
<td>2110C500</td>
<td></td>
</tr>
</tbody>
</table>

CEEHMMAH (DSA address 2110C340):

TRANSITION DSA

Saved Registers:

<table>
<thead>
<tr>
<th>GPR0....</th>
<th>2110C340</th>
<th>GPR1....</th>
<th>2110C340</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR2....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR3....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR4....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR5....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR6....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR7....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR8....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR9....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR10....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR11....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR12....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR13....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR14....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
<tr>
<td>GPR15....</td>
<td>00000000</td>
<td>2110C340</td>
<td></td>
</tr>
</tbody>
</table>
Handling dumps written to the z/OS UNIX file system

When a z/OS UNIX C/C++ application program is running in an address space created as a result of a call to spawn(), vfork(), or one of the exec family of functions, the SYSMDUMP DD allocation information is not inherited. Even though the SYSMDUMP allocation is not inherited, a SYSMDUMP allocation must exist in the parent in order to obtain a HFS storage dump.

Alternatively, you can specify the DYNDUMP runtime option to generate a system dump. For more information, see z/OS Language Environment Programming Reference.

If the program terminates abnormally while running in this new address space, the kernel causes an unformatted storage dump to be written to an HFS file in the user’s working directory. The file is placed in the current working directory or into /tmp if the current working directory is not defined. The file name has the following format; directory is the current working directory or tmp, and pid is the hexadecimal process ID (PID) for the process that terminated. For details on how to generate the system dump, see “Steps for generating a system dump in a z/OS UNIX shell” on page 88.

To debug the dump, use the MVS Interactive Problem Control System (IPCS). If the dump was written to an HFS file, you must allocate a data set that is large enough and has the correct attributes for receiving a copy of the HFS file. For example, from the ISPF DATA SET UTILITY panel you can specify a volume serial and data set name to allocate. Doing so brings up the DATA SET INFORMATION panel for specifying characteristics of the data set to be allocated. The following filled-in panel shows the characteristics defined for the URCOMPJRUSL.COREDUMP dump data set:
Fill in the information for your data set as shown, and estimate the number of cylinders required for the dump file you are going to copy.

Use the TSO/E OGET or OCOPY command with the BINARY keyword to copy the file into the data set. For example, to copy the HFS storage dump file coredump.00060007 into the data set URCOMP.JRUSL.COREDUMP just allocated, a user with the user ID URCOMP enters the following command:

```
OGET '/u/urcomp/coredump.00060007' 'urcomp.jrusl.coredump' BINARY
```

For more information on using the copy commands, see z/OS UNIX System Services User’s Guide.

After you have copied the storage dump file to the data set, you can use IPCS to analyze the dump. See “Formatting and analyzing system dumps” on page 89 for information about formatting Language Environment control blocks.

### Multithreading consideration

Certain control blocks are locked while a dump is in progress. For example, a csnap() of the file control block would prevent another thread from using or dumping the same information. An attempt to do so causes the second thread to wait until the first one completes before it can continue.

### Understanding C/C++ heap information in storage reports

Storage reports that contain specific C/C++ heap information can be generated in two ways; details on how to request and interpret the reports are provided in the following sections.

- By setting the Language Environment RPTSTG(ON) runtime option for Language Environment created heaps
- By issuing a stand-alone call to the C function __uheapreport() for user-created heaps.
Language Environment storage report with heap pools statistics

To request a Language Environment storage report set RPTSTG(ON). If the C/C++ application specified the HEAPPOOLS(ON) runtime option, then the storage report displays heap pools statistics. Figure 4 on page 17 is a sample storage report that shows heap pools statistics for a multithreaded C/C++ application. The following sections describe the C/C++ specific heap pools information.

HEAPPOOLS storage statistics

The HEAPPOOLS runtime option controls usage of the heap pools storage algorithm at the enclave level. The heap pools algorithm allows for the definition of one to twelve heap pools, each consisting of a number of storage cells of a specified length.

Note: The use of an alternative Vendor Heap Manager (VHM) overrides the use of the HEAPPOOLS runtime option.

HEAPPOOLS statistics:

• Pool p size: ssss Get requests: gggg
  
  $p$ the number of the pool. When there are multiple pools for a cell size, the pools are numbered using the format $aa.bbb$.
  
  $aa$ the number for the cell size.
  
  $bbb$ the number for the pool within the cell size.
  
  $ssss$ the cell size specified for the pool.
  
  $gggg$ the number of storage requests that were satisfied from this pool.

• Successful Get Heap requests: xxxx-yyyy $n$
  
  $xxxx$ the low side of the 8 byte range
  
  $yyyy$ the high side of the 8 byte range
  
  $n$ the number of requests in the 8 byte range.

• Requests greater than the largest cell size — the number of storage requests that are not satisfied by heap pools.

Note: Values displayed in the HEAPPOOLS statistics report are not serialized when collected; therefore, the values are not necessarily exact.

HEAPPOOLS summary: The HEAPPOOLS summary displays a report of the HEAPPOOLS statistics and provides suggested percentages for current cell sizes as well as suggested cell sizes.

• Specified Cell Size — the size of the cell specified in the HEAPPOOLS runtime option

• Element Size — the size of the cell plus any additional storage needed for control information or to maintain alignment

• Extent Percent — the cell pool percent specified by the HEAPPOOLS runtime option

• Cells Per Extent — the number of cells per extent. This number is calculated using the following formula, with a minimum of four cells:
  
  $\text{Initial Heap Size} \times (\text{Extent Percent}/100) / (\text{Element Size})$
Note: Having a small number of cells per extent is not recommended since the pool could allocate many extents, which would cause the HEAPPOOLS algorithm to perform inefficiently.

- Extents Allocated — the number of times that each pool allocated an extent. To optimize storage usage, the extents allocated should be either one or two. If the number of extents allocated is too high, then increase the percentage for the pool.
- Maximum Cells Used — the maximum number of cells used for each pool.
- Cells In Use — the number of cells that were never freed.

A large number in this field could indicate a storage leak.

- Suggested Percentages for current Cell Sizes — percentages calculated to find the optimal size of the cell pool extent. The calculation is based on the following formula:

\[
\text{Percentage} = \frac{(\text{Maximum Cells Used} \times \text{Element Size}) \times 100}{\text{Initial Heap Size}}
\]

With a minimum of 1% and a maximum of 90%

Make sure that your cell pool extents are neither too large nor too small. If your percentages are too large then additional, unreferenced virtual storage will be allocated, thereby causing the program to exhaust the region size. If the percentages are too small then the HEAPPOOLS algorithm will run inefficiently.

- Suggested Cell Sizes — sizes that are calculated to optimally use storage (assuming that the application will \_\_malloc/\_\_free with the same frequency). The suggested cell sizes are given with no percentages because the usage of each new cell pool size is not known. If there are less than 12 cell sizes calculated and the last calculated cell size is smaller than the largest cell size currently in effect, the largest cell size currently in effect will be used for the last suggested cell size.

For more information about stack and heap storage, see z/OS Language Environment Programming Guide.

C function \_\_uheapreport() storage report

To generate a user-created heap storage report use the C function, \_\_uheapreport(). Use the information in the report to assist with tuning your application's use of the user-created heap.

Figure 85 on page 230 shows a sample storage report generated by \_\_uheapreport().

For more information on the \_\_uheapreport() function, see z/OS XL C/C++ Runtime Library Reference. For tuning tips, see z/OS Language Environment Programming Guide.
User-created HeapPools statistics

- **Pool p size:** *ssss*
  - *p* the number of the pool
  - *ssss* the cell size specified for the pool.

- **Successful Get Heap requests:** *xxxx-yyyy n*
  - *xxxx* the low side of the range
  - *yyyy* the high side of the range
  - *n* the number of requests in the range.

- Requests greater than the largest cell size — the number of storage requests that are not satisfied by heap pools.

**Note:** Values displayed in the HeapPools statistics report are not serialized when collected; therefore, the values are not necessarily exact.

HeapPools summary

The HeapPools summary displays a report of the HeapPool statistics and provides suggested percentages for current cell sizes as well as suggested cell sizes.

- **Cell Size** — the size of the cell specified on the __ucreate() call
- **Extent Percent** — the cell pool percent specified on the __ucreate() call
- **Cells Per Extent** — the number of cells per extent. This number is calculated using the following formula:

  \[
  \text{Initial Heap Size} \times \left(\frac{\text{Extent Percent}}{100}\right) \div (8 + \text{Cell Size})
  \]

  with a minimum of four cells.
• Extents Allocated — the number of times that each pool allocated an extent.
• Maximum Cells Used — the maximum number of cells used for each pool.
• Cells In Use — the number of cells that were never freed.

Note: A large number in this field could indicate a storage leak.

• Suggested Percentages for current Cell Sizes — percentages calculated to find the optimal size of the cell pool extent. The calculation is based on the following formula:

\[(\text{Maximum Cells Used} \times (\text{Cell Size} + 8) \times 100) / \text{Initial Heap Size}\]

With a minimum of 1% and a maximum of 90%

Make sure that your cell pool extents are neither too large nor too small. If your percentages are too large then additional, unreferenced virtual storage will be allocated, thereby causing the program to exhaust the region size. If the percentages are too small then the HeapPools algorithm will run inefficiently.

• Suggested Cell Sizes — sizes that are calculated to optimally use storage (assuming that the application will \_umalloc/_ufree with the same frequency).

Note: The suggested cell sizes are given with no percentages because the usage of each new cell pool size is not known. If there are less than 12 cell sizes calculated and the last calculated cell size is smaller than the largest cell size currently in effect, the largest cell size currently in effect will be used for the last suggested cell size.

For more information about stack and heap storage, see _Z/OS Language Environment Programming Guide_

**MEMCHECK VHM memory leak analysis tool**

The MEMCHECK VHM memory leak analysis tool is an alternative vendor heap manager used to diagnose memory problems. MEMCHECK VHM performs the following functions and displays the results in two reports:

• check for heap storage leaks, double free, and overlays
• trace user heap storage allocation and deallocation requests

**Restrictions**

• MEMCHECK VHM works with C/C++ and Enterprise PL/I applications, but is not enabled for COBOL or Fortran.
• MEMCHECK VHM and HEAPPOOLS are mutually exclusive. HEAPPOOLS will be ignored when MEMCHECK VHM is active.
• MEMCHECK VHM should not be used in PIPI, PICI, CICS, and SPC environments.

**Invoking MEMCHECK VHM**

As with any alternate vendor heap manager, you must specify the dllname with the environment variable _CEE_HEAP_MANAGER to indicate that MEMCHECK VHM will be used to manage the user heap. Since CEE_HEAP_MANAGER must be set before any user code gains control, use the ENVAR runtime option to set the variable or set it inside the file specified by environment variables _CEE_ENVFILE or _CEE_ENVFILE_S. The format follows:
The following two DLLs are associated with MEMCHECK VHM and use the following events.

- CEL4MCHK: 31-bit base and XPLINK
- CELQMCHK: 64-bit

_VHM_INIT
replaces C-RTL malloc(), calloc(), realloc(), and free() with the corresponding MEMCHECK VHM functions. This event is only at Language Environment Initialization and only called by Language Environment.

_VHM_TERM
terminates Vendor Heap Manager to free the memcheck storage functions. This event is called only by Language Environment at Language Environment Termination.

_VHM_REPORT
generates the Heap Leak Report and the optional Trace Report. This new event will be called by Language Environment at Language Environment Termination and will write the Heap Leak Report (and the optional Trace Report if the _CEE_MEMCHECK_TRACE environment variable is active) in the output file name specified in _CEE_MEMCHECK_OUTFILENAME. This event can also be called dynamically by the __vhm_event() API.

MEMCHECK VHM environment variables
The MEMCHECK VHM environment variables control the tool, the call levels of the Heap Leak Report and Trace Report, the Overlay Analysis, the pad length added in the user heap allocation for overlay analysis, and the output file name for the reports. They should be activated through the ENVAR runtime option, the file specified by the _CEE_ENVFILE (or _CEE_ENVFILE_S) environment variable, or using the export command from the z/OS UNIX shell before any user code gets control (prior to the HLL user exit, static constructs, or main getting control). Setting these environment variables after the user code has begun execution will not activate them and the default values will be used.

_CEE_MEMCHECK_DEPTH
Description: Controls the number of call-levels to be generated on the Heap Leak Report.
Valid settings: integer value : the minimum is 1 and the maximum is 100. If the value specified is not valid, the default will be used.
Default: 10.

_CEE_MEMCHECK_OVERLAY
Description: Activates the storage overlays analysis beyond the end of the malloc’d storage.
Valid settings: ON to activate the analysis, OFF to deactivate. If an invalid value is specified, the default value will be used.
Default: OFF

_CEE_MEMCHECK_OVERLAYLEN
Description: Sets the pad length added in the user heap allocation for overlay analysis. This environment variable will be used only if _CEE_MEMCHECK_OVERLAY is active.
Valid settings: integer value, multiple of 8: the minimum is 8 and the maximum is 80. Non-multiples of 8 will be rounded up to the next multiple.

Default: 8

_CEE_MEMCHECK_TRACE
Description: Enables tracing of all heap storage allocation and deallocation and a Trace Report will be generated at Language Environment Termination.

Valid settings: ON to activate the analysis, OFF to deactivate. If an invalid value is specified, the default value will be used.

Default: OFF

_CEE_MEMTRACE_DEPTH
Description: Controls the number of call-levels to be generated in the Trace Report, on each call to a library function that deals with heap. This environment variable will be used only if _CEE_MEMCHECK_TRACE is active.

Valid settings: integer value: the minimum is 1 and the maximum is 100. If the value specified is not valid, the default value will be used.

Default: 10

_CEE_MEMCHECK_OUTFILENAME
Description: Sets the name of the fully qualified path name of the file in which the Heap Leak Report and Trace Report should be directed. The report name could be any valid name used in C-RTL fopen() function, then it could also generates the reports in a Data Set.

Valid settings: string value. If an invalid value is specified, the default value will be used.

Default: standard error output

MEMCHECK VHM report sample scenario
In this example, the MEMCHECK VHM tool is used by specifying the environment variables from the z/OS UNIX shell. The user specifies a depth of 8 call levels in the Heap Leak Report and 8 call levels in the Trace Report for 31-bit.

1. Specifies the depth to trace on storage requests (written to the Heap Leak Report):
   Export _CEE_MEMCHECK_DEPTH=8

2. Activates the Trace Report option:
   Export _CEE_MEMCHECK_TRACE=ON

3. Specifies the depth to trace on storage requests (written to the Trace Report):
   Export _CEE_MEMTRACE_DEPTH=8

4. Activates the Overlay analysis option:
   Export _CEE_MEMCHECK_OVERLAY=ON

5. Activates the tool with the 31-bit DLL (automatically generating the Heap Leak Report):
   Export _CEE_HEAP_MANAGER=CEL4MCHK

MEMCHECK VHM report examples
Both reports are written at Language Environment termination (_VHM_TERM event). They are written in the output file name specified in _CEE_MEMCHECK_OUTFILENAME and are consistent with the format of other Language Environment reports.
The following trace report will be generated at Language Environment termination (_VHM_TERM event) if the _CEE_MEMCHECK_TRACE environment variable is active. The report generates the traceback information of all heap storage allocations and deallocations.

MEMCHECK
Language Environment V1 R7
TRACE REPORT for enclave main, termination report

DEALLOCATE of storage at 0x25a2ea30
- sequence 12
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 2576f888 +000002b0 _cterm
  Called from: 05046788 +0000040c (unknown)

DEALLOCATE of storage at 0x25a2edc8
- sequence 11
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 2576f888 +000001b0 _cterm
  Called from: 05046788 +0000040c (unknown)

DEALLOCATE of storage at 0x25a2ecf8
- sequence 10
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 25601ae8 +00000020 function3
  Called from: 25601bb8 +00000080 function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000062 main

ALLOCATE of storage at 0x25a2ecf8 for 5 bytes
- sequence 9
  Called from: 25a44330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 25601ae8 +00000020 function3
  Called from: 25601bb8 +00000080 function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000062 main

ALLOCATE of storage at 0x25a2ecd8 for 8 bytes
- sequence 8
  Called from: 25a44330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 25601ae8 +00000020 function3
  Called from: 25601bb8 +00000080 function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000062 main

DEALLOCATE of storage at 0x25a2edc8
- sequence 7
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 25601ae8 +00000020 function3
  Called from: 25601bb8 +00000080 function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000062 main

DEALLOCATE of storage at 0x25a2ecd8
- sequence 6
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 25601ae8 +00000020 function3
  Called from: 25601bb8 +00000080 function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000062 main

ALLOCATE of storage at 0x25a2ecd8 for 4 bytes
- sequence 5
  Called from: 25a44330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 25601ae8 +00000020 function3
  Called from: 25601bb8 +00000080 function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000062 main

ALLOCATE of storage at 0x25a2ec90 for 48 bytes
- sequence 4
  Called from: 25a44330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 25725c08 +000000a0 dllinit
  Called from: 05049c88 +000007dc (unknown)

ALLOCATE of storage at 0x25a2ea30 for 504 bytes
- sequence 3
  Called from: 25a44330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPITF
  Called from: 258c6670 +00000180 setlocale
  Called from: 25862540 +00000590 tzset
  Called from: 257f8d30 +00002df2 _cinit
  Called from: 0504abb0 +00000c40 (unknown)

ALLOCATE of storage at 0x25a2e1f8 for 2074 bytes
- sequence 2

---
The Heap Leak Report (Figure 86) will be generated with any remaining entries in the memory leak control block. The allocated entries will be reported as storage leaks, while the deallocated entries will be reported as duplicated deallocations and the overlay entries as overlay damage.

Figure 86. Heap Leak Report generated by MEMCHECK VHM
The following names are used within MEMCHECK to denote special cases and may be displayed in any of the reports:

(unknown)
Name of the routine is not known.

(nome)
Routine does not have a name in the PPA section. (For example, module compiled with compress option).

(nospace)
Internal memory space reserved by MEMCHECK is full, so name was not saved for the traceback information. No action is needed from the user.
Chapter 5. Debugging COBOL programs

This section provides information for debugging applications that contain one or more COBOL programs. It includes information about:

- Determining the source of error
- Generating COBOL listings and the Language Environment dump
- Finding COBOL information in a dump
- Debugging example COBOL programs

Determining the source of error

The following sections describe how you can determine the source of error in your COBOL program. They explain how to simplify the process of debugging COBOL programs by using features such as the DISPLAY statement, declaratives, and file status keys. The following methods for determining errors are covered:

- Tracing program logic
- Finding and handling input/output errors
- Validating data
- Assessing switch problems
- Generating information about procedures

After you have located and fixed any problems in your program, you should delete all debugging aids and recompile it before running it in production. Doing so helps the program run more efficiently and use less storage.

For detailed information about any of the topics and techniques discussed in the following sections, refer to the appropriate COBOL documentation in the Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733).

Tracing program logic

You can add DISPLAY statements to help you trace through the logic of the program in a non-CICS environment. If, for example, you determine that the problem appears in an EVALUATE statement or in a set of nested IF statements, DISPLAY statements in each path tell you how the logic flows. You can also use DISPLAY statements to show you the value of interim results. Scope terminators can also help you trace the logic of your program because they clearly indicate the end of a statement.

For example, to check logic flow, you might insert the following statement to determine if you started and finished a particular procedure:

```
DISPLAY "ENTER CHECK PROCEDURE".
   .
   . (checking procedure routine)
   .
DISPLAY "FINISHED CHECK PROCEDURE".
```

After you are sure that the program works correctly, comment out the DISPLAY statement lines by putting asterisks in position 7 of the appropriate lines.
Finding input/output errors

VSAM file status keys can help you determine whether routine errors are due to the logic of your routine or are I/O errors occurring on the storage media. To use file status keys as a debugging aid, include a test after each I/O statement to check for a value other than 0 in the file status key. If the value is other than 0, you can expect to receive an error message. You can use a nonzero value to indicate how the I/O procedures in the routine were coded. You can also include procedures to correct the error based on the file status key value.

Handling input/output errors

If you have determined that the problem lies in one of the I/O procedures in your program, you can include the USE EXCEPTION/ERROR declarative to help debug the problem. If the file does not open, the appropriate USE EXCEPTION/ERROR declarative is activated. You can specify the appropriate declarative for the file or for the different open attributes: INPUT, OUTPUT, I/O, or EXTEND. Code each USE AFTER STANDARD ERROR statement in a separate section immediately after the Declarative Section keyword of the Procedure Division.

Validating data (class test)

If you suspect that your program is trying to perform arithmetic on nonnumeric data or is somehow receiving the wrong type of data on an input record, you can use the class test to validate the type of data.

Assessing switch problems

Using INITIALIZE or SET statements to initialize a table or data item is useful when you suspect that a problem is caused by residual data left in those fields. If your problem occurs intermittently and not always with the same data, the problem could be that a switch is not initialized, but is generally set to the right value (0 or 1). By including a SET statement to ensure that the switch is initialized, you can determine if the uninitialized switch is the cause of the problem.

Generating information about procedures

You can use the USE FOR DEBUGGING declarative to include COBOL statements in a COBOL program and specify when they should run. Use these statements to generate information about your program and how it is running. Code each USE FOR DEBUGGING declarative in a separate section in the DECLARATIVES SECTION of the PROCEDURE DIVISION.

For example, to check how many times a procedure is run, include a special procedure for debugging (in the USE FOR DEBUGGING declarative) that adds 1 to a counter each time control passes to that procedure. The adding-to-a-counter technique can be used as a check for:

- How many times a PERFORM ran. This shows you whether the control flow you are using is correct.
- How many times a loop routine actually runs. This tells you whether the loop is running and whether the number you have used for the loop is accurate.

You can use debugging lines, debugging statements, or both in your program. Debugging lines are placed in your program, and are identified by a D in position 7. Debugging statements are coded in the DECLARATIVES SECTION of the PROCEDURE DIVISION.

- The USE FOR DEBUGGING declaratives must:
- Be only in the DECLARATIVES SECTION
- Follow a DECLARATIVES header USE FOR DEBUGGING

With USE FOR DEBUGGING, the TEST compiler option must have the NONE hook-location suboption specified or the NOTEST compiler option must be specified. The TEST compiler option and the DEBUG runtime option are mutually exclusive, with DEBUG taking precedence.

- Debugging lines must have a 0 in position 7 to identify them.

To use debugging lines and statements in your declarative procedures, you must include both:
- WITH DEBUGGING MODE in the SOURCE-COMPUTER paragraph in the ENVIRONMENT DIVISION
- The DEBUG runtime option

Figure 87 shows how to use the DISPLAY statement and the USE FOR DEBUGGING declarative to debug a program.

```
Environment Division
Source Computer . . . With Debugging Mode.

: Data Division.

: File Section.
   Working-Storage Section.
   *(among other entries you would need:)
   01 Trace-Msg PIC X(30) Value "Trace for Procedure-Name: ".
   01 Total PIC 9(9) Value Zeros.
   *(balance of Working-Storage Section)

Procedure Division.
Declaratives.
Debug-Decl Section.
   Use For Debugging On 501-Some-Routine.
Debug-Decl-Paragraph.
   Display Trace-Msg, Debug-Name, Total.
Debug-Decl-End.
   Exit.
End Declaratives.
Begin-Program Section.

: Perform 501-Some-Routine.
   *(within the module where you want to test, place:)
   Add 1 To Total
   *(whether you put a period at the end depends on
   * where you put this statement.)
```

Figure 87. Example of using the WITH DEBUGGING MODE clause

In the example in Figure 87, portions of a program are shown to illustrate the kind of statements needed to use the USE FOR DEBUGGING declarative. The DISPLAY statement specified in the DECLARATIVES SECTION issues the following message
every time the PERFORM 501-SOME-ROUTINE runs. The total shown, \( nn \), is the value accumulated in the data item named TOTAL:

\[
\text{Trace For Procedure-Name : 501-Some-Routine} \quad nn
\]

Another use for the DISPLAY statement technique shown above is to show the flow through your program. You do this by changing the USE FOR DEBUGGING declarative in the DECLARATIVES SECTION to the following value and dropping the word TOTAL from the DISPLAY statement.

\[
\text{USE FOR DEBUGGING ON ALL PROCEDURES.}
\]

### Using COBOL listings

When you are debugging, you can use one or more of the listings shown in Table 40. The following sections give an overview of each of these listings and the compiler option you use to obtain each listing.

**Table 40. Compiler-generated COBOL listings and their contents**

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted Cross-Reference Listings</td>
<td>Provides sorted cross-reference listings of DATA DIVISION, PROCEDURE DIVISION, and program names. The listings provide the location of all references to this information.</td>
<td>XREF</td>
</tr>
<tr>
<td>Data Map listing</td>
<td>Provides information about the locations of all DATA DIVISION items and all implicitly declared variables. This option also supplies a nested program map, which indicates where the programs are defined and provides program attribute information.</td>
<td>MAP</td>
</tr>
<tr>
<td>Verb Cross-Reference listing</td>
<td>Produces an alphabetic listing of all the verbs in your program and indicates where each is referenced.</td>
<td>VBREF</td>
</tr>
<tr>
<td>Procedure Division listings</td>
<td>Tells the COBOL compiler to generate a listing of the PROCEDURE DIVISION along with the assembler coding produced by the compiler. The list output includes the assembler source code, a map of the task global table (TGT), information about the location and size of WORKING-STORAGE and control blocks, and information about the location of literals and code for dynamic storage usage.</td>
<td>LIST</td>
</tr>
<tr>
<td>Procedure Division listings</td>
<td>Instead of the full PROCEDURE DIVISION listing with assembler expansion information, you can use the OFFSET compiler option to get a condensed listing that provides information about the program verb usage, global tables, WORKING-STORAGE, and literals. The OFFSET option takes precedence over the LIST option. That is, OFFSET and LIST are mutually exclusive; if you specify both, only OFFSET takes effect.</td>
<td>OFFSET</td>
</tr>
</tbody>
</table>

### Generating a Language Environment dump of a COBOL program

The sample programs shown in Figure 88 on page 241 and Figure 89 on page 242 generate Language Environment dumps with COBOL-specific information.

**COBOL program that calls another COBOL program**

In Figure 88 on page 241 program COBDUMP1 calls COBDUMP2, which in turn calls the Language Environment dump service CEE3DMP.
COBOL program that calls the Language Environment CEE3DMP callable service

In the example in Figure 89 on page 242, program COBDUMP2 calls the Language Environment dump service CEE3DMP.
Sample Language Environment dump with COBOL-specific information

The call in program COBDUMP2 to CEE3DMP generates a Language Environment dump, shown below. The dump includes a traceback section, which shows the names of both programs, a section on register usage at the time the dump was generated, and a variables section, which shows the storage and data items for each program. Note that the high half of register 14 at entry to CEE3DMP is not available and is shown in the dump as ********. Character fields in the dump are indicated by single quotes. For an explanation of these sections of the dump, see "Finding COBOL information in a dump" on page 244.
Finding COBOL information in a dump

Like the standard Language Environment dump format, dumps generated from COBOL programs contain:

- Control block information for active programs
- Storage for each active program
- Enclave-level data
- Process-level data

Control block information for active routines

The Control Blocks for Active Routines section of the dump, shown in Figure 90, displays the following information for each active COBOL program:

- DSA
- Program name and date/time of compile
- COBOL compiler Version, Release, Modification, and User Level
- COBOL compiler Options
- COBOL control blocks TGT and CLLE. The layout of the TGT can be found by looking at the compiler listing of the COBOL program. For Enterprise COBOL V5.1 and later releases, the TGT is replaced by COBDSACB. The CLLE is a COBOL control block that is allocated by the COBOL runtime for each program. The CLLE is dumped for IBM service personnel use.

![Figure 90. Control block information for active COBOL routines](image-url)
Storage for each active routine

The Storage for Active Routines section of the dump, shown in “Storage for active COBOL programs,” displays the following information for each COBOL program:

- Program name
- Contents of the base locators for files, WORKING-STORAGE, LINKAGE SECTION, LOCAL-STORAGE SECTION, variably-located areas, and EXTERNAL data.
- File record contents.
- WORKING-STORAGE, including the base locator for WORKING-STORAGE (BLW) and program class storage.

Storage for active COBOL programs:

<table>
<thead>
<tr>
<th>Storage for Active Routines: COBUMP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents of base locators for files are:</td>
</tr>
<tr>
<td>- +000000 0001E338</td>
</tr>
<tr>
<td>Contents of base locators for WORKING-STORAGE are:</td>
</tr>
<tr>
<td>- B-1144A108</td>
</tr>
<tr>
<td>Contents of base locators for the LINKAGE SECTION are:</td>
</tr>
<tr>
<td>- 0-00000000 1-1144A120</td>
</tr>
<tr>
<td>No indexes were used in this program.</td>
</tr>
<tr>
<td>No variably-located areas were used in this program.</td>
</tr>
<tr>
<td>No EXTERNAL data was used in this program.</td>
</tr>
<tr>
<td>No OBJECT instance data were used in this program.</td>
</tr>
<tr>
<td>No LOCAL-STORAGE was used in this program.</td>
</tr>
<tr>
<td>No DSA indexes were used in this program.</td>
</tr>
<tr>
<td>No FACTORY data was used in this program.</td>
</tr>
<tr>
<td>No XML data was used in this program.</td>
</tr>
</tbody>
</table>

File record contents for COBUMP2

ESDS1DD DUMPBUF (BIL-0): 0011E838
- +000000 0001E338 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ........................................ |
- +000000 0011E338 - +000000 0001E338 |

WORKING-STORAGE for COBUMP2

BLW: 1144A120
- +000000 1144A108 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ...XXX...YYY...ZZZ...COBOL DUMP |
- +000000 1144A108 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ............SYSDUMP ............ |
- +000000 1144A108 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ..XXX...ZZZ...COBOL DUMP |
- +000000 1144A108 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |

LINKAGE SECTION for COBUMP2

BLW: 1144A120
- +000000 1144A120 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
- +000000 1144A120 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ............SYSDUMP ............ |
- +000000 1144A120 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ..XXX...ZZZ...COBOL DUMP |
- +000000 1144A120 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |

Program class storage: 1144A140
- +000000 1144A140 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
- +000000 1144A140 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ............SYSDUMP ............ |
- +000000 1144A140 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ..XXX...ZZZ...COBOL DUMP |
- +000000 1144A140 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |

Chapter 5. Debugging COBOL programs 245
Enclave-level data

The Enclave Control Blocks section of the dump, shown in Figure 91 on page 247, displays the following information:

- RUNCOM control block. The RUNCOM is a control block that is allocated by the COBOL runtime to anchor enclave level resources. The RUNCOM is dumped for IBM service personnel use.

  **Note:** In Enterprise COBOL V5.1 and later releases, the RUNCOM control block is replaced by the COBEDB control block.

- Storage for all run units

- COBOL control blocks FCB, FIB, and GMAREA. The FCB, FIB, and GMAREA are control blocks used for COBOL file processing. These control blocks are dumped for IBM service personnel use.
Figure 91. Enclave-level data for COBOL programs
Process-level data

The Process Control Block section of the dump, shown in Figure 92, displays COBOL process-level control blocks THDCOM, COBCOM, COBVEC, and ITBLK. For Enterprise COBOL V5.1 and later releases, the process-level control blocks are COBPCB (corresponds to THDCOM), COBRCB (corresponds to COBCOM) and LIBVEC (corresponds to COBVEC). The control blocks are dumped for IBM service personnel use. In a non-CICS environment, the ITBLK control block only appears when a VS COBOL II program is active. In a CICS environment, the ITBLK control block always appears.

Debugging example COBOL programs

The following examples help demonstrate techniques for debugging COBOL programs. Important areas of the dump output are highlighted. Data unnecessary to debugging has been replaced by vertical ellipses.

Subscript range error

Figure 93 on page 249 illustrates the error of using a subscript value outside the range of an array. This program was compiled with LIST, TEST, and SSRANGE. The SSRANGE compiler option causes the compiler to generate code that checks (during run time) for data that has been stored or referenced outside of its defined area because of incorrect indexing and subscripting. The SSRANGE option takes effect during run time. For COBOL V4R2 and prior releases, you can disable the check by specifying the CHECK(OFF) runtime option. For Enterprise COBOL V5.1 and later releases, the CHECK runtime option is ignored.

The program was run with TERMTHDACT(TRACE) to generate the traceback information shown in “Sections of Language Environment dump for COBOLX” on page 249.
To understand the traceback information and debug this program, use the following steps:

1. Locate the current error message in the Condition Information for Active Routines section of the Language Environment traceback, shown in “Sections of Language Environment dump for COBOLX.” The message is IGZ0006S The reference to table SLOT by verb number 01 on line 000011 addressed an area outside the region of the table. The message indicates that line 11 was the current COBOL statement when the error occurred. For more information about this message, see z/OS Language Environment Runtime Messages.

2. Statement 11 in the traceback section of the dump occurred in program COBOLX.

3. Find the statement on line 11 in the listing for program COBOLX, shown in Figure 94. This statement moves the 1 value to the array SLOT (J).

4. Find the values of the local variables in the Parameters, Registers, and Variables for Active Routines section of the traceback, shown in “Sections of Language Environment dump for COBOLX.” J, which is of type PIC 9(4) with usage COMP, has a 9 value. J is the index to the array SLOT.

The array SLOT contains eight positions. When the program tries to move a value into the J or 9th element of the 8-element array named SLOT, the error of moving a value outside the area of the array occurs.

Sections of Language Environment dump for COBOLX

/* COBOLX
  000001  ID DIVISION.
  000002  PROGRAM-ID. COBOLX.
  000003  ENVIRONMENT DIVISION.
  000005  WORKING-STORAGE SECTION.
  000006  77 J PIC 9(4) USAGE COMP.
  000007  01 TABLE-X.
  000008  02 SLOT PIC 9(4) USAGE COMP OCCURS 8 TIMES.
  000009  PROCEDURE DIVISION.
  000010  MOVE 9 TO J.  6
  000011  MOVE 1 TO SLOT (J).  8 6
  000012  GOBACK.
*/ COBOLX

Figure 94. COBOL listing for COBOLX

Chapter 5. Debugging COBOL programs 249
**Calling a nonexistent subroutine**

Figure 95 on page 251 demonstrates the error of calling a nonexistent subroutine in a COBOL program. In this example, the program COBOLY was compiled with the compiler options LIST, MAP and XREF. The TEST option was also specified with the suboptions NONE and SYM. Figure 95 on page 251 shows the program.
To understand the traceback information and debug this program, use the following steps:

1. **Locate the error message for the original condition under the Condition Information for Active Routines section of the dump, shown in Figure 96 on page 252.** The message is CEE3501S The module UNKNOWN was not found. For more information about this message, see z/OS Language Environment Runtime Messages.

2. **Note the sequence of calls in the Traceback section of the dump.** COBOL called IGZCFCC; IGZCFCC (a COBOL library subroutine used for dynamic calls) called IGZCLDL; then IGZCLDL (a COBOL library subroutine used to load library routines) called CEESGLT, a Language Environment condition handling routine.

   This sequence indicates that the exception occurred in IGZCLDL when COBOL was attempting to make a dynamic call. The call statement in COBOL is located at offset +0000036E.

   **Note:** If COBOL is compiled with Enterprise COBOL V5.1 or later releases, the traceback section of Language Environment dump is shown in Figure 97 on page 252. The only difference is that IGZCFCC and IGZCLDL are combined and replaced by IGZXFCA1. (IGZXFCA1 attempts to load a non-existent routine.)

---

```cobol
CBL LIST,MAP,XREF,TEST(NONE,SYM)
ID DIVISION.
PROGRAM-ID. COBOLY.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 SUBNAME PIC X(8) USAGE DISPLAY VALUE 'UNKNOWN'.
PROCEDURE DIVISION.
   CALL SUBNAME.
   GOBACK.
```

*Figure 95. COBOL example of calling a nonexistent subroutine*
3. Use the offset of 'X'36E' from the COBOL listing, shown below, to locate the statement that caused the exception in the COBOL program. At offset 'X'36E' is an instruction for statement 8. Statement 8 is a call with the identifier SUBNAME specified.

**Figure 96. Sections of Language Environment dump for COBOL**

<table>
<thead>
<tr>
<th>Traceback:</th>
<th>DSA Entry</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CEEHDSP</td>
<td>+00004030</td>
<td></td>
<td></td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CEEHSLGT</td>
<td>+0000085C</td>
<td></td>
<td></td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IGZCLDL</td>
<td>+0000012A</td>
<td></td>
<td></td>
<td>IGZCPAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IGZCFCC</td>
<td>+000003EE</td>
<td></td>
<td></td>
<td>IGZCPAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>COBOL</td>
<td>+0000036E</td>
<td>B</td>
<td>GO</td>
<td>COBOL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DSA Addr</th>
<th>E Addr</th>
<th>PU Addr</th>
<th>PU Offset</th>
<th>Comp Date</th>
<th>Compile Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1147E6F0</td>
<td>1128023B</td>
<td>+00004030</td>
<td>20061215</td>
<td>CEL</td>
</tr>
<tr>
<td>2</td>
<td>1147E558</td>
<td>112CF960</td>
<td>+0000085C</td>
<td>20061215</td>
<td>CEL</td>
</tr>
<tr>
<td>3</td>
<td>1147E3B0</td>
<td>11453680</td>
<td>+0000012A</td>
<td>20061213</td>
<td>LIBRARY</td>
</tr>
<tr>
<td>4</td>
<td>1147E1D0</td>
<td>1140A5F8</td>
<td>+000003EE</td>
<td>20061213</td>
<td>LIBRARY</td>
</tr>
<tr>
<td>5</td>
<td>1147E030</td>
<td>000083F0</td>
<td>+0000036E</td>
<td>20070214</td>
<td>COBOL</td>
</tr>
</tbody>
</table>

**Figure 97. Portion of traceback of Language Environment dump for COBOL (when compiled with Enterprise COBOL V5.1 or later)**

<table>
<thead>
<tr>
<th>Traceback:</th>
<th>DSA Entry</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CEEHDSP</td>
<td>+00004220</td>
<td></td>
<td></td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CEEHSLGT</td>
<td>+00000060</td>
<td></td>
<td></td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IGZCFCA1</td>
<td>+00000930</td>
<td></td>
<td></td>
<td>IGZLPKPA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>COBOL</td>
<td>+000001CA</td>
<td>B</td>
<td>GO</td>
<td>COBOL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DSA Addr</th>
<th>E Addr</th>
<th>PU Addr</th>
<th>PU Offset</th>
<th>Comp Date</th>
<th>Compile Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2605E6E8</td>
<td>2506E6E8</td>
<td>+00004220</td>
<td>20110318</td>
<td>CEL</td>
</tr>
<tr>
<td>2</td>
<td>2605E490</td>
<td>250F9578</td>
<td>+00000060</td>
<td>29110318</td>
<td>CEL</td>
</tr>
<tr>
<td>3</td>
<td>2605E240</td>
<td>25F648C0</td>
<td>+00000030</td>
<td>20130605</td>
<td>LIBRARY</td>
</tr>
<tr>
<td>4</td>
<td>2605E030</td>
<td>25000000</td>
<td>+0000001CA</td>
<td>20130607</td>
<td>COBOLV5+ EBCDIC HFP</td>
</tr>
</tbody>
</table>

3. Use the offset of 'X'36E' from the COBOL listing, shown below, to locate the statement that caused the exception in the COBOL program. At offset 'X'36E' is an instruction for statement 8. Statement 8 is a call with the identifier SUBNAME specified.

```plaintext
PP 5655-G53 IBM Enterprise COBOL for z/OS 3.4.1 COBOLY Date 02/14/2007 Time 14:59:07 Page 11
000280 START EQU *
000280 183F LR 3,15
000282 4100 11A0 LA 0,416(0,1)
```
IBM Enterprise COBOL for z/OS 3.4.1

Chapter 5. Debugging COBOL programs
4. Find the value of the local variables in the Parameters, Registers, and Variables for Active Routines section of the dump, shown in Figure 98. Notice that the value of SUBNAME with usage DISP, has a value of 'UNKNOWN'.

Correct the problem by either changing the subroutine name to one that is defined, or by ensuring that the subroutine is available at compile time.

---

**Figure 98. Parameters, registers, and variables for active routines section of dump for COBOL**

**Divide-by-zero error**

The following example demonstrates the error of calling an assembler routine that tries to divide by zero. Both programs were compiled with TEST(STMT_SYM) and run with the TERMTHDACT(TRACE) runtime option. Figure 99 on page 255 shows the main COBOL program (COBOLZ1), the COBOL subroutine (COBOLZ2), and the assembler routine.
To debug this application, use the following steps:

1. Locate the error message for the current condition in the Condition Information section of the dump, shown in “Sections of Language Environment dump for program COBOLZ1” on page 257. The message is CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).

For additional information about this message, see z/OS Language Environment Runtime Messages.

2. Note the sequence of calls in the call chain. COBOLZ1 called IGZCFCC, which is a COBOL library subroutine used for dynamic calls; IGZCFCC called COBOLZ2; COBOLZ2 then called IGZCFCC; and IGZCFCC called ASSEMZ3. The exception occurred at this point, resulting in a call to CEEHDSP, a Language Environment condition handling routine.

The call to ASSEMZ3 occurred at statement 11 of COBOLZ2. The exception occurred at offset +64 in ASSEMZ3.

---

[Main Program]

```
CBL TEST(STMT,SYM),DYN,XREF(FULL),MAP
ID DIVISION.
  PROGRAM-ID. COBOLZ1.
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
    77 D-VAL PIC 9(4) USAGE COMP VALUE 0.
  PROCEDURE DIVISION.
    CALL "COBOLZ2" USING D-VAL.
    GOBACK.
```

[Subroutine]

```
CBL TEST(STMT,SYM),DYN,XREF(FULL),MAP
ID DIVISION.
  PROGRAM-ID. COBOLZ2.
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
    77 D-VAL PIC 9(4) USAGE COMP.
  LINKAGE SECTION.
    77 D-VAL PIC 9(4) USAGE COMP.
  PROCEDURE DIVISION USING D-VAL.
    MOVE D-VAL TO DV-VAL.
    CALL "ASSEMZ3" USING DV-VAL.
    GOBACK.
```

[Assembler Routine]

```
PRINT NOGEN
ASSEMZ3 CEEENTRY MAIN=NO,PPA=MAINPPA
  LA 5,234B  Low order part of quotient
  SR 4,4    Hi order part of quotient
  L 6,0(1)  Get pointer to divisor
  LA 6,0(6) Clear hi bit
  D 4,0(6)  Do division
  CEETERM RC=0 Terminate with return code zero

* MAINPPA CEEPPA Constants describing the code block
  CEEDA Mapping of the Dynamic Save Area
  CEECAA Mapping of the Common Anchor Area
  END ASSEMZ3
```

Figure 99. Main COBOL program, COBOL subroutine, and assembler routine

To debug this application, use the following steps:

1. Locate the error message for the current condition in the Condition Information section of the dump, shown in “Sections of Language Environment dump for program COBOLZ1” on page 257. The message is CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).

For additional information about this message, see z/OS Language Environment Runtime Messages.

2. Note the sequence of calls in the call chain. COBOLZ1 called IGZCFCC, which is a COBOL library subroutine used for dynamic calls; IGZCFCC called COBOLZ2; COBOLZ2 then called IGZCFCC; and IGZCFCC called ASSEMZ3. The exception occurred at this point, resulting in a call to CEEHDSP, a Language Environment condition handling routine.

The call to ASSEMZ3 occurred at statement 11 of COBOLZ2. The exception occurred at offset +64 in ASSEMZ3.
3. Locate statement 11 in the COBOL listing for the COBOLZ2 program, shown in Figure 100. This is a call to the assembler routine ASSEMZ3.

```
/* COBOLZ2 */
000001  ID DIVISION.
000002  PROGRAM-ID. COBOLZ2.
000003  ENVIRONMENT DIVISION.
000004  DATA DIVISION.
000005  WORKING-STORAGE SECTION.
000006  77 DV-VAL PIC 9(4) USAGE COMP.
000007  LINKAGE SECTION.
000008  77 D-VAL PIC 9(4) USAGE COMP.
000009  PROCEDURE DIVISION USING D-VAL.
000010  MOVE D-VAL TO DV-VAL.
000011  CALL "ASSEMZ3" USING DV-VAL.
000012  GOBACK.
+/ COBOLZ2
```

**Figure 100. COBOL listing for COBOLZ2**

4. Check offset +64 in the listing for the assembler routine ASSEMZ3, shown in Figure 101. This shows an instruction to divide the contents of register 4 by the variable pointed to by register 6. You can see the two instructions preceding the divide instruction load register 6 from the first word pointed to by register 1 and prepare register 6 for the divide. Because of linkage conventions, you can infer that register 1 contains a pointer to a parameter list that passed to ASSEMZ3. Register 6 points to a 0 value because that was the value passed to ASSEMZ3 when it was called by a higher level routine.

```
000056  4150 092C 38 LA 5,2348 Low order part of quotient
000056  38B4 39 SR 4,4 Hi order part of quotient
000060  4166 0000 41 LA 6,0(1) Get pointer to divisor
000064  504D 0000 42 LA 6,0(6) Clear hi bit
000068  5BF0 00C8 43 CEETERM RC=0 Terminate with return code zero
```

**Figure 101. Listing for ASSEMZ3**

5. Check local variables for COBOLZ2 in the Local Variables section of the dump shown in Figure 102 on page 257. From the dump and listings, you know that...
COBOLZ2 called ASSEMZ3 and passed a parameter in the variable DV-VAL. The two variables DV-VAL and D-VAL have 0 values.

6. In the COBOLZ2 subroutine, the variable D-VAL is moved to DV-VAL, the parameter passed to the assembler routine. D-VAL appears in the Linkage section of the COBOLZ2 listing, shown in Figure 103, indicating that the value did pass from COBOLZ1 to COBOLZ2.

7. In the Local Variables section of the dump for program COBOLZ1, shown in Figure 104, D-VAL has a 0 value. This indicates that the error causing a fixed-point divide exception in ASSEMZ3 was actually caused by the value of D-VAL in COBOLZ1.
Condition Information for Active Routines

COBOLZ2 (DSA address 1147E750):

Saved Registers:

GPR0....... 1147E570 GPR1....... 1147E4C0 GPR2....... 1147E348 GPR3....... 1147E2B0 GPR4....... 1147E198 GPR5....... 1147E068 GPR6....... 1147E350 GPR7....... 1147E220 GPR8....... 1147E190 GPR9....... 1147E060 GPR10..... 1147E228 GPR11..... 1147E048 GPR12..... 1147E210 GPR13..... 1147E030 GPR14..... 1147E188 GPR15..... 1147E018 GPR16..... 1147E170 GPR17..... 1147E020 GPR18..... 1147E160 GPR19..... 1147E200 GPR20..... 1147E018 GPR21..... 1147E180 GPR22..... 1147E000 GPR23..... 1147E150 GPR24..... 1147E218 GPR25..... 1147E038 GPR26..... 1147E178 GPR27..... 1147E028 GPR28..... 1147E168 GPR29..... 1147E208 GPR30..... 1147E018 GPR31..... 1147E188

GPR STORAGE:

+0200 1147E570 1147E4C0 1147E348 1147E2B0 1147E198 1147E068 1147E350 1147E220 1147E190 1147E060 1147E228 1147E048 1147E210 1147E030 1147E188 1147E018 1147E170 1147E020 1147E160 1147E200 1147E018 1147E180 1147E000 1147E150 1147E218 1147E038 1147E178 1147E028 1147E168 1147E208 1147E018 1147E188

Local Variables:

6 77 D-VAL 9999 COMP 0000
8 77 D-VAL 9999 COMP 0000

COBOLZ1 (DSA address 1147E030):

Saved Registers:

GPR0....... 1147E110 GPR1....... 1147E128 GPR2....... 1147E088 GPR3....... 1147E060 GPR4....... 1147E040 GPR5....... 1147E020 GPR6....... 1147E000 GPR7....... 1147E180 GPR8....... 1147E160 GPR9....... 1147E140 GPR10..... 1147E120 GPR11..... 1147E100 GPR12..... 1147E080 GPR13..... 1147E060 GPR14..... 1147E040 GPR15..... 1147E020 GPR16..... 1147E000 GPR17..... 1147E180 GPR18..... 1147E160 GPR19..... 1147E140 GPR20..... 1147E120 GPR21..... 1147E100 GPR22..... 1147E080 GPR23..... 1147E060 GPR24..... 1147E040 GPR25..... 1147E020 GPR26..... 1147E000 GPR27..... 1147E180 GPR28..... 1147E160 GPR29..... 1147E140 GPR30..... 1147E120 GPR31..... 1147E100

GPR STORAGE:

+0200 1147E110 1147E128 1147E088 1147E060 1147E040 1147E020 1147E000 1147E180 1147E160 1147E140 1147E120 1147E100 1147E080 1147E060 1147E040 1147E020 1147E000 1147E180 1147E160 1147E140 1147E120 1147E100 1147E080 1147E060 1147E040 1147E020 1147E000 1147E180 1147E160 1147E140 1147E120 1147E100 1147E080 1147E060 1147E040 1147E020 1147E000

Local Variables:

6 77 D-VAL 9999 COMP 0000
Chapter 6. Debugging Fortran routines

This section provides information to help you debug applications that contain one or more Fortran routines. It includes the following topics:

- Determining the source of errors in Fortran routines
- Using Fortran compiler listings
- Generating a Language Environment dump of a Fortran routine
- Finding Fortran information in a dump
- Examples of debugging Fortran routines

Determining the source of errors in Fortran routines

Most errors in Fortran routines can be identified by the information provided in Fortran runtime messages, which begin with the prefix “FOR”. The Fortran compiler cannot identify all possible errors. The following list identifies several errors not detected by the compiler that could potentially result in problems:

- Failing to assign values to variables and arrays before using them in your program.
- Specifying subscript values that are not within the bounds of an array. If you assign data outside the array bounds, you can inadvertently destroy data and instructions.
- Moving data into an item that is too small for it, resulting in truncation.
- Making invalid data references to EQUIVALENCE items of differing types (for example, integer or real).
- Transferring control into the range of a DO loop from outside the range of the loop. The compiler issues a warning message for all such branches if you specify OPT(2), OPT(3), or VECTOR.
- Using arithmetic variables and constants that are too small to give the precision you need in the result. For example, to obtain more than 6 decimal digits in floating-point results, you must use double precision.
- Concatenating character strings in such a way that overlap can occur.
- Trying to access services that are not available in the operating system or hardware.
- Failing to resolve name conflicts between Fortran and C library routines using the procedures described in "z/OS Language Environment Programming Guide".

Identifying runtime errors

Fortran has several features that help you find runtime errors. Fortran runtime messages are discussed in "z/OS Language Environment Runtime Messages". Other debugging aids include the optional traceback map, program interruption messages, abnormal termination dumps, and operator messages.

- The optional traceback map helps you identify where errors occurred while running your application. The TERMTHDACT(TRACE) runtime option, which is set by default under Language Environment, generates a dump containing the traceback map.

You can also get a traceback map at any point in your routine by invoking the ERRTRA subroutine.
Program interruption messages are generated whenever the program is interrupted during execution. Program interruption messages are written to the Language Environment message file.

The program interruption message indicates the exception that caused the termination; the completion code from the system indicates the specification or operation exception resulting in termination.

Program interruptions causing an abnormal termination produce a dump, which displays the completion code and the contents of registers and system control fields.

To display the contents of main storage as well, you must request an abnormal termination (ABEND) dump by including a SYSUDUMP DD statement in the appropriate job step. The following example shows how the statement can be specified for IBM-supplied cataloged procedures:

```
//GO.SYSUDUMP DD SYSOUT=A
```

You can request various dumps by invoking any of several dump service routines while your program runs. These dump service routines are discussed in “Generating a Language Environment dump of a Fortran routine” on page 261.

Operator messages are displayed when your program issues a PAUSE or STOP n statement. These messages help you understand how far execution has progressed before reaching the PAUSE or STOP statement.

The operator message can take the following forms:

- **n** String of 1–5 decimal digits you specified in the PAUSE or STOP statement. For the STOP statement, this number is placed in R15.
- **‘message’** Character constant you specified in the PAUSE or STOP statement.
- **0** Printed when a PAUSE statement containing no characters is executed (not printed for a STOP statement).

A PAUSE message causes the program to stop running pending an operator response. The format of the operator’s response to the message depends on the system being used.

Under Language Environment, error messages produced by Language Environment and Fortran are written to a common message file. Its ddname is specified in the MSGFILE runtime option. The default ddname is SYSOUT.

Fortran information directed to the message file includes:

- Error messages resulting from unhandled conditions
- Printed output from any of the dump services (SDUMP, DUMP/PDUMP, CDUMP/CPDUMP)
- Output produced by a WRITE statement with a unit identifier having the same value as the Fortran error message unit
- Output produced by a WRITE statement with * given as the unit identifier (assuming the Fortran error message unit and standard print unit are the same unit)
- Output produced by the PRINT statement (assuming the Fortran error message unit and the standard print unit are the same unit)

For more information about handling message output using the Language Environment MSGFILE runtime option, see [z/OS Language Environment Programming Guide](#).
Using Fortran compiler listings

Fortran listings provide you with:
- The date of compilation including information about the compiler
- A listing of your source program
- Diagnostic messages telling you of errors in the source program
- Informative messages telling you the status of the compilation

Table 41 lists of the contents of the various compiler-generated listings that you might find helpful when you use information in dumps to debug Fortran programs.

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostic message listing</td>
<td>Error messages detected during compilation.</td>
<td>FLAG</td>
</tr>
<tr>
<td>Source program</td>
<td>Source program statements.</td>
<td>SOURCE</td>
</tr>
<tr>
<td>Source program</td>
<td>Source program statements and error messages.</td>
<td>SRCFLG</td>
</tr>
<tr>
<td>Storage map and cross reference</td>
<td>Variable use, statement function, subprogram, or intrinsic function within a program.</td>
<td>MAP and XREF</td>
</tr>
<tr>
<td>Cross reference</td>
<td>Cross reference of names with attributes.</td>
<td>XREF</td>
</tr>
<tr>
<td>Source program map</td>
<td>Offsets of automatic and static internal variables (from their defining base).</td>
<td>MAP</td>
</tr>
<tr>
<td>Object code</td>
<td>Contents of the program control section in hexadecimal notation and translated into a pseudo-assembler format. To limit the size of the object code listing, specify the statement or range of statements to be listed; for example, LIST(20) or LIST(10,30).</td>
<td>LIST</td>
</tr>
<tr>
<td>Variable map, object code, static storage</td>
<td>Same as MAP and LIST options above, plus contents of static internal and static external control sections in hexadecimal notation with comments.</td>
<td>MAP and LIST</td>
</tr>
<tr>
<td>Symbolic dump</td>
<td>Internal statement numbers, sequence numbers, and symbol (variable) information.</td>
<td>SDUMP</td>
</tr>
</tbody>
</table>

Generating a Language Environment dump of a Fortran routine

To generate a dump containing Fortran information, call either DUMP/PDUMP, CDUMP/CPDUMP, or SDUMP. DUMP/PDUMP and CDUMP/CPDUMP produce output that is unchanged from the output generated under Fortran. Under Language Environment, however, the output is directed to the message file.

When SDUMP is invoked, the output is also directed to the Language Environment message file. The dump format differs from other Fortran dumps, however, reflecting a common format shared by the various HLLs under Language Environment.

You cannot make a direct call to CEE3DMP from a Fortran program. It is possible to call CEE3DMP through an assembler routine called by your Fortran program. Fortran programs are currently restricted from directly invoking Language Environment callable services.
DUMP/PDUMP
Provides a dump of a specified area of storage.

CDUMP/CPDUMP
Provides a dump of a specified area of storage in character format.

SDUMP
Provides a dump of all variables in a program unit.

DUMP/PDUMP subroutines
The DUMP/PDUMP subroutine dynamically dumps a specified area of storage to the system output data set. When you use DUMP, the processing stops after the dump; when you use PDUMP, the processing continues after the dump.

Syntax

CALL {DUMP | PDUMP} (a1, b1,k1, a2,b2, k2,...)

a and b
Variables in the program unit. Each indicates an area of storage to be dumped. Either a or b can represent the upper or lower limit of the storage area.

k
The dump format to be used. The values that can be specified for k, and the resulting dump formats, are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Format Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Hexadecimal</td>
</tr>
<tr>
<td>1</td>
<td>LOGICAL*1</td>
</tr>
<tr>
<td>2</td>
<td>LOGICAL*4</td>
</tr>
<tr>
<td>3</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>5</td>
<td>REAL*4</td>
</tr>
<tr>
<td>6</td>
<td>REAL*8</td>
</tr>
<tr>
<td>7</td>
<td>COMPLEX*8</td>
</tr>
<tr>
<td>8</td>
<td>COMPLEX*16</td>
</tr>
<tr>
<td>9</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>10</td>
<td>REAL*16</td>
</tr>
<tr>
<td>11</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td>12</td>
<td>UNSIGNED*1</td>
</tr>
<tr>
<td>13</td>
<td>INTEGER*1</td>
</tr>
<tr>
<td>14</td>
<td>LOGICAL*2</td>
</tr>
<tr>
<td>15</td>
<td>INTEGER*8</td>
</tr>
<tr>
<td>16</td>
<td>LOGICAL*8</td>
</tr>
</tbody>
</table>

Usage considerations for DUMP/PDUMP
A load module or phase can occupy a different area of storage each time it is executed. To ensure that the appropriate areas of storage are dumped, the following conventions should be observed.

If an array and a variable are to be dumped at the same time, a separate set of arguments should be used for the array and for the variable. The specification of limits for the array should be from the first element in the array to the last element. For example, assume that A is a variable in common, B is a real number, and TABLE is an array of 20 elements. The following call to the storage dump routine could be used to dump TABLE and B in hexadecimal format, and stop the program after the dump is taken.

CALL DUMP(TABLE(1),TABLE(20),0,B,B,0)
If an area of storage in common is to be dumped at the same time as an area of storage not in common, the arguments for the area in common should be given separately. For example, the following call to the storage dump routine could be used to dump the variables A and B in REAL*8 format without stopping the program.

```
CALL PDUMP(A,A,6,B,B,6)
```

If variables not in common are to be dumped, each variable must be listed separately in the argument list. For example, if R, P, and Q are defined implicitly in the program, the following statement should be used to dump the three variables in REAL*4 format.

```
CALL PDUMP(R,R,5,P,P,5,Q,Q,5)
```

If the following statement is used, all main storage between R and Q is dumped, which might or might not include P, and could include other variables.

```
CALL PDUMP(R,Q,5)
```

### CDUMP/CPDUMP subroutines

The CDUMP/CPDUMP subroutine dynamically dumps a specified area of storage containing character data. When you use CDUMP, the processing stops after the dump; when you use CPDUMP, the processing continues after the dump.

#### Syntax

```
CALL {CDUMP | CPDUMP} (a1, b1, a2, b2,...)
```

- **a** and **b**
  - Variables in the program unit. Each indicates an area of storage to be dumped.
  - Either a or b can represent the upper or lower limit of each storage area.

The dump is always produced in character format. A dump format type (unlike for DUMP/PDUMP) must not be specified.

#### Usage considerations for CDUMP/CPDUMP

A load module can occupy a different area of storage each time it is executed. To ensure that the appropriate areas of storage are dumped, the following conventions should be observed.

If an array and a variable are to be dumped at the same time, a separate set of arguments should be used for the array and for the variable. The specification of limits for the array should be from the first element in the array to the last element. For example, assume that B is a character variable and TABLE is a character array of 20 elements. The following call to the storage dump routine could be used to dump TABLE and B in character format, and stop the program after the dump is taken.

```
CALL CDUMP(TABLE(1), TABLE(20), B, B)
```

### SDUMP subroutine

The SDUMP subroutine provides a symbolic dump that is displayed in a format dictated by variable type as coded or defaulted in your source. Data is dumped to the error message unit. The symbolic dump is created by program request, on a program unit basis, using CALL SDUMP. Variables can be dumped automatically after abnormal termination using the compiler option SDUMP. For more information on the SDUMP compiler option, see *VS FORTRAN Version 2 Programming Guide for CMS and MVS.*
Items displayed are:

- All referenced, local, named, and saved variables in their Fortran-defined data representation
- All variables contained in a static common area (blank or named) in their Fortran-defined data representation
- All variables contained in a dynamic common area in their Fortran-defined data representation
- Nonzero or nonblank character array elements only
- Array elements with their correct indexes

The amount of output produced can be very large, especially if your program has large arrays, or large arrays in common blocks. For such programs, you might want to avoid calling SDUMP.

**Syntax**

```call call sdump [(rtn1, rtn2,...)]

rtn1, rtn2,...

Names of other program units from which data will be dumped. These names must be listed in an EXTERNAL statement.
```

**Usage considerations for SDUMP**

- To obtain symbolic dump information and location of error information, compilation must be done either with the SDUMP option or with the TEST option.
- Calling SDUMP and specifying program units that have not been entered gives unpredictable results.
- Calling SDUMP with no parameters produces the symbolic dump for the current program unit.
- An EXTERNAL statement must be used to identify the names being passed to SDUMP as external routine names.
- At higher levels of optimization (1, 2, or 3), the symbolic dump could show incorrect values for some variables because of compiler optimization techniques.
- Values for uninitialized variables are unpredictable. Arguments in uncalled subprograms or in subprograms with argument lists shorter than the maximum can cause the SDUMP subroutine to fail.
- The display of data can also be invoked automatically. If the runtime option TERMTHDACT(DUMP) is in effect and your program abends in a program unit compiled with the SDUMP option or with the TEST option, all data in that program unit is automatically dumped. All data in any program unit in the save area traceback chain compiled with the SDUMP option or with the TEST option is also dumped. Data occurring in a common block is dumped at each occurrence, because the data definition in each program unit could be different.

Examples of calling SDUMP from the main program and from a subprogram follow. Figure 105 on page 265 shows a sample program calling SDUMP and Figure 107 on page 266 shows the resulting output that is generated. In the main program, the following statement

```
EXTERNAL PGM1, PGM2, PGM3
```

makes the address of subprograms PGM1, PGM2, and PGM3 available for a call to SDUMP, as follows:
CALL SDUMP (PGM1, PGM2, PGM3)

This causes variables in PGM1, PGM2, and PGM3 to be printed.

In the subprogram PGM1, the following statement makes PGM2 and PGM3 available. (PGM1 is missing because the call is in PGM1.)

EXTERNAL PGM2, PGM3

The following statements dump the variables PGM1, PGM2, and PGM3.

CALL SDUMP
CALL SDUMP (PGM2, PGM3)

Figure 105. Example program that calls SDUMP
Figure 107 shows the resulting output generated by the example in Figure 105 on page 265.

Finding Fortran information in a Language Environment dump

To locate Fortran-specific information in a Language Environment dump, you must understand how to use the traceback section and the section in the symbol table dump showing parameters and variables. Figure 108 on page 267 shows an example of a Fortran dump; Table 42 on page 267 provides additional information.
Figure 108. Sections of the Language Environment dump

Table 42 describes the sections shown in the sample dump in Figure 108

Table 42. Understanding the Language Environment traceback table

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>The traceback section of the dump contains condition information about your routine and information about the statement number and address where the exception occurred. The traceback section helps you locate where an error occurred in your program. The information in this section begins with the most recent program unit and ends with the first program unit.</td>
</tr>
<tr>
<td>[2]</td>
<td>The condition information section contains information for the active routines. It indicates the program message, program unit name, the statement number, and the offset within the program unit where the error occurred.</td>
</tr>
</tbody>
</table>
Table 42. Understanding the Language Environment traceback table  (continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3]</td>
<td>The local variable section contains information on all variables and arrays in each program unit in the save area chain, including the program that caused the dump to be invoked. The output shows variable items (one line only) and array (more than one line) items. Use the local variable section of the dump to identify the variable name, type, and value at the time the dump was called. Variable and array items can contain either character or noncharacter data, but not both.</td>
</tr>
<tr>
<td>[4]</td>
<td>The file status and attribute section of the dump displays the total number of units defined, the default units for error messages, and the default unit numbers for formatted input or formatted output.</td>
</tr>
</tbody>
</table>

Examples of debugging Fortran routines

This section contains examples of Fortran routines and instructions for using information in the Language Environment dump to debug them.

Calling a nonexistent routine

Figure 109 illustrates an error caused by calling a nonexistent routine. The options in effect at compile time appear at the top of the listing.

```
OPTIONS IN EFFECT: LIST NOMAP NOXREF NOQOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NODIM NORENT SDUMP(ISN)
   NOSYM NOVECTOR IL(DIM) NOST TEST SC(+) NOCC NODE NODECA NOICA NODIRECTIVE NODBCS NODC NODEC NODEMP NODYNAM NOSYM NOREORDER NOPC
   OPT(0) LANGLVL(77) NOFIPS FLAG(I) HALT(S) AUTODBL(NONE) PTRSIZE(8) LINCOUNT(60) CHARLEN(500) NAME(MAIN#)
1  PROGRAM CALLNON
2       CALL SUBNAM
3  STOP
5  END
```

Figure 109. Example of calling a nonexistent routine

Figure 110 on page 269 shows sections of the dump generated by a call to SDUMP.
Information for enclave CALLNON

Information for thread 0000000000000000

Traceback:

<table>
<thead>
<tr>
<th>DSA Addr</th>
<th>Program Unit</th>
<th>PU Addr</th>
<th>PU Offset</th>
<th>Entry</th>
<th>E Addr</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Service Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002D018</td>
<td>CEEHOSP</td>
<td>05936760</td>
<td>+000027C</td>
<td>CEEHOSP</td>
<td>05936760</td>
<td>+000027C</td>
<td>CEEPLPKA</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>05900C10</td>
<td>CALLNON</td>
<td>05900B28</td>
<td>-05900B26</td>
<td>CALLNON</td>
<td>05900B28</td>
<td>-05900B26</td>
<td>3_ISN</td>
<td>GO</td>
<td>Exception</td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Condition Information for CALLNON (DSA address 05900C10)

CIB Address: 0002D468

Current Condition:
CEE3201S The system detected an operation exception.

Location:
Program Unit: CALLNON
Entry: CALLNON
Statement: 3_ISN Offset: -05900B26

Machine State:
ILC..... 0002 Interruption Code..... 0001
PSW..... 07FD5300 08000000
GPR0..... F0000000 GPR1..... 00000000 GPR2..... 05900D04 GPR3..... 05900C10
GPR4..... 007F63D0 GPR5..... 007FD3F8 GPR6..... 007F67F8 GPR7..... FD000000
GPR8..... 007FD908 GPR9..... 007FD968 GPR10..... 007F6930 GPR11..... 00E21ED2
GPR12..... 002E1100 GPR13..... 05900C10 GPR14..... 85900CE8 GPR15..... 00000000

Storage dump near condition, beginning at location: 00000000
+00000 00000000 Inaccessible storage.

Parameters, Registers, and Variables for Active Routines:

CEEHOSP (DSA address 0002D018):

Saved Registers:
GPR0..... 00000000 GPR1..... 00000000 GPR2..... 0002D007 GPR3..... 0002E027
GPR4..... 0002DF94 GPR5..... 00000000 GPR6..... 00000000 GPR7..... 00000000
GPR8..... 0002D7E9 GPR9..... 0593875E GPR10..... 0593775F GPR11..... 05936760
GPR12..... 00014770 GPR13..... 0002D018 GPR14..... 8002500E GPR15..... 85949C70

GREG STORAGE:
Storage around GPR0 (00000000)
+00000 00000000 Inaccessible storage.
+00000 00000020 Inaccessible storage.
+00000 00000040 Inaccessible storage.

Storage around GPR1 (0002D3B4)
-00000 0002D3D0 00000006 00000000 0002D017 0593875E 0593775F 05936760 00014770 00000000
...........lg:1..~1........
+00000 0002D3D0 0002D3D0 00000000 00000000 00000000 00000007 859067E0 00000000 00591448
...........m...m...m..d..d..d..d....

: :

Figure 110. Sections of the Language Environment dump resulting from a call to a nonexistent routine

To understand the traceback section, and debug this example routine, do the following:

1. Find the Current Condition information in the Condition Information for Active Routines section of the dump. The message is CEE3201S. The system detected an operation exception at statement 3. For more information about this message, see [z/OS Language Environment Runtime Messages](https://www.ibm.com/support/knowledgecenter/SSDUXK_1.5.0/com.ibm.zos.v1r5.SCZCH00/zos_ioc.ep_004.html). This section of the dump also provides such information as the name of the active routine and the current statement number at the time of the dump.

2. Locate statement 3 in the routine shown in [Figure 109 on page 268](#). This statement calls subroutine SUBNAM. The message CEE3201S in the Condition Information section of the dump indicates that the operation exception was generated because of an unresolved external reference.

3. Check the linkage editor output for error messages.

File Status and Attributes:
The total number of units defined is 100.
The default unit for the PUNCH statement is 7.
The default unit for the Fortran error messages is 6.
The default unit for formatted sequential output is 6.
The default unit for formatted sequential input is 5.
Divide-by-zero error

Figure 111 demonstrates a divide-by-zero error. In this example, the main Fortran program passed 0 to subroutine DIVZEROSUB, and the error occurred when DIVZEROSUB attempted to use this data as a divisor.

```
OPTIONS IN EFFECT: LIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NODDIM NORENT SDUMP(ISN)
NOSYM NOVECTOR IL(DIM) NOTEST SC(+) NODC NOEC NODECK NOICA NODIRECTIVE NODBCS NOSAA NOPARALLEL NODYNAMIC NOSYM
NOREORDER NOPC
OPT(0) LANG_LVL(77) NOFIPS FLAG(I) HALT(S) AUTODBL(NONE) PRTSIZE(8) LINECOUNT(60) CHARLEN(500) NAME(MAIN#)

1 7 PROGRAM DIVZERO
2 8 INTEGER*4 ANY_NUMBER
3 9 INTEGER*4 ANY_ARRAY(3)
4 10 PRINT *, 'EXAMPLE STARTING'
5 11 DO I = 1,3
6 12 ANY_ARRAY(I) = I
7 13 END DO
8 14 CALL DIVZEROSUB(ANY_NUMBER, ANY_ARRAY)
9 15 PRINT *, 'EXAMPLE ENDING'
10 16 STOP
11 17 END

OPTIONS IN EFFECT: LIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NODDIM NORENT SDUMP(ISN)
NOSYM NOVECTOR IL(DIM) NOTEST SC(+) NODC NOEC NODECK NOICA NODIRECTIVE NODBCS NOSAA NOPARALLEL NODYNAMIC NOSYM
NOREORDER NOPC
OPT(0) LANG_LVL(77) NOFIPS FLAG(I) HALT(S) AUTODBL(NONE) PRTSIZE(8) LINECOUNT(60) CHARLEN(500) NAME(MAIN#)

1 3 SUBROUTINE DIVZEROSUB(DIVISOR, DIVIDEND)
2 4 INTEGER*4 DIVISOR
3 5 INTEGER*4 DIVIDEND(3)
4 6 PRINT *, 'IN SUBROUTINE DIVZEROSUB'
5 7 DIVIDEND(3) = DIVIDEND(3) / DIVISOR
6 8 PRINT *, 'END OF SUBROUTINE DIVZEROSUB'
7 9 RETURN
8 10 END
```

Figure 111. Fortran routine with a divide-by-zero error

Figure 112 on page 271 shows the Language Environment dump for routine DIVZERO.
Figure 112. Language Environment dump from divide-by-zero Fortran example

To debug this application, do the following:

1. Locate the error message, CEE3209S, for the current condition in the Condition Information section of the dump, shown in Figure 112. The system detected a fixed-point divide exception. See z/OS Language Environment Runtime Messages for additional information about this message.

2. Note the sequence of the calls in the call chain:
   a. DIVZERO called AFHLCLNR, which is a Fortran library subroutine.

Chapter 6. Debugging Fortran routines 271
b. AFHLCLNR called DIVZEROSUB.

Note: When a program-unit name is longer than 7 characters, the name as it appears in the dump consists of the first 4 and last 3 characters concatenated together.

c. DIVZEROSUB attempted a divide-by-zero operation at statement 5.

d. This resulted in a call to CEEHDSP, a Language Environment condition handling routine.

3. Locate statement 5 in the Fortran listing for the DIVZEROSUB subroutine in Figure 112 on page 271. This is an instruction to divide the contents of DIVIDEND(3) by DIVISOR.

4. Since DIVISOR is a parameter of subroutine DIVZEROSUB, go to the Parameters section of the dump shown in Figure 112 on page 271. The parameter DIVISOR shows a value of 0.

5. Since DIVISOR contains the value passed to DIVZEROSUB, check its value. ANY_NUMBER is the actual argument passed to DIVZEROSUB, and the dump and listing of DIVZERO indicate that ANY_NUMBER had value 0 when passed to DIVZEROSUB, leading to the divide-by-zero exception.
Chapter 7. Debugging PL/I for MVS & VM routines

This section contains information that can help you debug applications that contain one or more PL/I for MVS & VM routines. Following a discussion about potential errors in PL/I for MVS & VM routines, the first topic discusses how to use compiler-generated listings to obtain information about PL/I for MVS & VM routines, and how to use PLIDUMP to generate a Language Environment dump of a PL/I for MVS & VM routine. The last part of this section provides examples of PL/I for MVS & VM routines and explains how to debug them using information contained in the traceback information provided in the dump.

Determining the source of errors in PL/I for MVS & VM routines

Most errors in PL/I for MVS & VM routines can be identified by the information provided in PL/I runtime messages, which begin with the prefix IBM. For a list of these messages, see z/OS Language Environment Runtime Messages.

A malfunction in running a PL/I for MVS & VM routine can be caused by:
- Logic errors in the source routine
- Invalid use of PL/I for MVS & VM
- Unforeseen errors
- Invalid input data
- Compiler or runtime routine malfunction
- System malfunction
- Unidentified routine malfunction
- Overlaid storage

Logic errors in the source routine

Errors of this type are often difficult to detect because they often appear as compiler or library malfunctions. Some common errors in source routines are:
- Incorrect conversion from arithmetic data
- Incorrect arithmetic and string manipulation operations
- Unmatched data lists and format lists

Invalid use of PL/I for MVS & VM

A misunderstanding of the language or a failure to provide the correct environment for using PL/I for MVS & VM can result in an apparent malfunction of a PL/I for MVS & VM routine. Any of the following, for example, might cause a malfunction:
- Using uninitialized variables
- Using controlled variables that have not been allocated
- Reading records into incorrect structures
- Misusing array subscripts
- Misusing pointer variables
- Incorrect conversion
- Incorrect arithmetic operations
- Incorrect string manipulation operations

Unforeseen errors

If an error is detected during run time and no ON-unit is provided in the routine to terminate the run or attempt recovery, the job terminates abnormally. However,
the status of a routine at the point where the error occurred can be recorded by using an ERROR ON-unit that contains the statements. In the following example, the statement ON ERROR SYSTEM ensures that further errors do not result in a permanent loop.

```
ON ERROR BEGIN;
  ON ERROR SYSTEM;
  CALL PLIDUMP;  /*generates a dump*/
  PUT DATA;     /*displays variables*/
END;
```

Invalid input data
A routine should contain checks to ensure that any incorrect input data is detected before it can cause the routine to malfunction. Use the COPY option of the GET statement to check values obtained by stream-oriented input. The values are listed on the file named in the COPY option. If no file name is given, SYSPRINT is assumed.

Compiler or runtime routine malfunction
If you are certain that the malfunction is caused by a compiler or runtime routine error, you can either open a PMR or submit an APAR for the error. For more information about handling compiler and runtime routine malfunctions, see the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735) Meanwhile, you can try an alternative way to perform the operation that is causing the trouble. A bypass is often feasible, since the PL/I for MVS & VM language frequently provides an alternative method of performing operations.

System malfunction
System malfunctions include machine malfunctions and operating system errors. System messages identify these malfunctions and errors to the operator.

Unidentified routine malfunction
In most circumstances, an unidentified routine malfunction does not occur when using the compiler. If your routine terminates abnormally without an accompanying Language Environment runtime diagnostic message, the error causing the termination might also be inhibiting the production of a message. Check for the following:
• Your job control statements might be in error, particularly in defining data sets.
• Your routine might overwrite main storage areas containing executable instructions. This can happen if you have accidentally:
  – Assigned a value to a nonexistent array element. For example:
    To detect this type of error in a compiled module, set the SUBSCRIPTRANGE

```
DCL ARRAY(10);

  DO I = 1 TO 100;
  ARRAY(I) = VALUE;
```

condition so that each attempt to access an element outside the declared range of subscript values raises the SUBSCRIPTRANGE condition. If there is no ON-unit for this condition, a diagnostic message is printed and the
ERROR condition is raised. This facility, though expensive in run time and storage space, is a valuable routine-testing aid.

- Used an incorrect locator value for a locator (pointer or offset) variable. This type of error can occur if a locator value is obtained by means of record-oriented transmission. Ensure that locator values created in one routine, transmitted to a data set, and subsequently retrieved for use in another routine, are valid for use in the second routine.

- Attempted to free a nonbased variable. This can happen when you free a based variable after its qualifying pointer value has been changed. For example:

```pli
DCL A STATIC,B BASED (P);
ALLOCATE B;
P = ADDR(A);
FREE B;
```

- Used the SUBSTR pseudovariable to assign a string to a location beyond the end of the target string. For example:

```pli
DCL X CHAR(3);
I=3
SUBSTR(X,2,I) = 'ABC';
```

To detect this type of error, enable the STRINGRANGE condition during compilation.

Storage overlay problems

If you suspect an error in your PL/I for MVS & VM application is a storage overlay problem, check for the following:

1. The use of a subscript outside the declared bounds (check the SUBSCRIPTRANGE condition)
2. An attempt to assign a string to a target with an insufficient maximum length (check the STRINGSIZE condition)
3. The failure of the arguments to a SUBSTR reference to comply with the rules described for the SUBSTR built-in function (check the STRINGRANGE condition)
4. The loss of significant last high-order (left-most) binary or decimal digits during assignment to an intermediate result or variable or during an input/output operation (check the SIZE condition)
5. The reading of a variable-length file into a variable
6. The misuse of a pointer variable
7. The invocation of a Language Environment callable service with fewer arguments than are required

The first four situations are associated with the listed PL/I for MVS & VM conditions, all of which are disabled by default. If you suspect one of these problems exists in your routine, use the appropriate condition prefix on the suspected statement or on the BEGIN or PROCEDURE statement of the containing block.
The fifth situation occurs when you read a data record into a variable that is too small. This type of problem only happens with variable-length files. You can often isolate the problem by examining the data in the file information and buffer.

The sixth situation occurs when you misuse a pointer variable. This type of storage overlay is particularly difficult to isolate. There are a number of ways pointer variables can be misused:

- When a READ statement runs with the SET option, a value is placed in a pointer. If you then run a WRITE statement or another READ SET option with another pointer, you overlay your storage if you try to use the original pointer.
- When you try to use a pointer to allocate storage that has already been freed, you can also cause a storage overlay.
- When you attempt to use a pointer set with the ADDR built-in function as a base for data with different attributes, you can cause a storage overlay.

The seventh situation occurs when a Language Environment callable service is passed fewer arguments than its interface requires. The following example might cause a storage overlay because Language Environment assumes that the fourth item in the argument list is the address of a feedback code, when in reality it could be residue data pointing anywhere in storage.

<table>
<thead>
<tr>
<th>Invalid calls</th>
<th>Valid calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL CEEDATE ENTRY OPTIONS(ASM); CALL CEEDATE(x,y,z); /* invalid */</td>
<td>DCL CEEDATE ENTRY(<em>,</em>,<em>,</em> OPTIONAL) OPTIONS(ASM); CALL CEEDATE(x,y,z,<em>); /</em> valid */</td>
</tr>
<tr>
<td>CALL CEEDATE(x,y,z); /* invalid */</td>
<td>CALL CEEDATE(x,y,z,fc); /* valid */</td>
</tr>
</tbody>
</table>

Using PL/I for MVS & VM compiler listings

The following sections explain how to generate listings that contain information about your routine. PL/I for MVS & VM listings show machine instructions, constants, and external or internal addresses that the linkage editor resolves. This information can help you find other information, such as variable values, in a dump of a PL/I for MVS & VM routine. The PL/I compiler listings included in the following sections are from the PL/I for MVS & VM product.

Generating PL/I for MVS & VM listings and maps

Table 43 shows compiler-generated listings that you might find helpful when you use information in dumps to debug PL/I for MVS & VM routines. For more information about supported compiler options that generate listings, reference the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735)

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source program</td>
<td>Source program statements</td>
<td>SOURCE</td>
</tr>
<tr>
<td>Cross reference</td>
<td>Cross reference of names with attributes</td>
<td>XREF and ATTRIBUTES</td>
</tr>
<tr>
<td>Aggregate table</td>
<td>Names and layouts of structures and arrays</td>
<td>AGGREGATE</td>
</tr>
<tr>
<td>Variable map</td>
<td>Offsets of automatic and static internal variables (from their defining base)</td>
<td>MAP</td>
</tr>
</tbody>
</table>
Table 43. Compiler-generated PL/I for MVS & VM listings and their contents (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object code</td>
<td>Contents of the program control section in hexadecimal notation and translated into a pseudo-assembler format. To limit the size of the object code listing, specify a certain statement or range of statements to be listed; for example, LIST(20) or LIST(10,30).</td>
<td>LIST</td>
</tr>
<tr>
<td>Variable map, object code, static storage</td>
<td>Same as MAP and LIST options, plus contents of static internal and static external control sections in hexadecimal notation with comments</td>
<td>MAP and LIST</td>
</tr>
</tbody>
</table>

Finding information in PL/I for MVS & VM listings

Figure 113 shows an example PL/I for MVS & VM routine that was compiled with LIST and MAP:

```pli
// PROCESS SOURCE, LIST, MAP;
// SOURCE LISTING
//
STMT
  1 | EXAMPLE: PROC OPTIONS(MAIN);
  2 |   DCL EXTR ENTRY EXTERNAL;
  3 |   DCL A FIXED BIN(31);
  4 |   DCL B(2,2) FIXED BIN(31) STATIC EXTERNAL INIT((4)0);
  5 |   DCL C CHAR(20) STATIC INIT('SAMPLE CONSTANT');
  6 |   DCL D FIXED BIN(31) STATIC;
  7 |   DCL E FIXED BIN(31);
  8 |   FETCH EXTR;
  9 |   CALL EXTR(A,B,C,D,E);
 10 |   DISPLAY(C);
11 |  END;
```

Figure 113. PL/I for MVS & VM routine compiled with LIST and MAP

Figure 114 on page 278 shows the output generated by the LIST and MAP options for this routine, including the static storage map, variable storage map, and the object code listing. The sections following this example describe the contents of each type of listing.
Figure 114. Compiler-generated listings from example PL/I for MVS & VM routine
Static internal storage map

To get a complete variable storage map and static storage map, but not a complete LIST, specify a single statement for LIST to minimize the size of the listing; for example, LIST(1).

Each line of the static storage map contains the following information:

1. Six-digit hexadecimal offset.
2. Hexadecimal text, in 8-byte sections where possible.
3. Comment, indicating the type of item to which the text refers. The comment appears on the first line of the text for an item. Table 44 lists some typical comments you might find in a static storage listing.

<table>
<thead>
<tr>
<th>Comment</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A..xxx</td>
<td>Address constant for xxx</td>
</tr>
<tr>
<td>COMPILER LABEL CL..n</td>
<td>Compiler-generated label n</td>
</tr>
<tr>
<td>CONDITION CSECT</td>
<td>Control section for programmer-named condition</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>Constant</td>
</tr>
<tr>
<td>CSECT FOR EXTERNAL VARIABLE</td>
<td>Control section for external variable</td>
</tr>
<tr>
<td>D..xxx</td>
<td>Descriptor for xxx</td>
</tr>
<tr>
<td>DED..xxx</td>
<td>Data element descriptor for xxx</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
<td>Data descriptor</td>
</tr>
<tr>
<td>ENVB</td>
<td>Environment control block</td>
</tr>
<tr>
<td>FECB..xxx</td>
<td>Fetch control block for xxx</td>
</tr>
<tr>
<td>DCLCB</td>
<td>Declare control block</td>
</tr>
<tr>
<td>FED..xxx</td>
<td>Format element descriptor for xxx</td>
</tr>
</tbody>
</table>

Table 44. Typical comments in a PL/I for MVS & VM static storage listing

Figure 115. Compiler-generated listings from example PL/I for MVS & VM routine (continued)
Table 44. Typical comments in a PL/I for MVS & VM static storage listing (continued)

<table>
<thead>
<tr>
<th>Comment</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>KD..xxx</td>
<td>Key descriptor for xxx</td>
</tr>
<tr>
<td>LOCATOR..xxx</td>
<td>Locator for xxx</td>
</tr>
<tr>
<td>ONCB</td>
<td>ON statement control block</td>
</tr>
<tr>
<td>PICTURED DED..xxx</td>
<td>Pictured data element descriptor for xxx</td>
</tr>
<tr>
<td>PROGRAM ACON</td>
<td>Program address constant</td>
</tr>
<tr>
<td>RD..xxx</td>
<td>Record descriptor for xxx</td>
</tr>
<tr>
<td>SYMBOL TABLE ELEMENT</td>
<td>Symbol table address</td>
</tr>
<tr>
<td>SYMBOL TABLE..xxx</td>
<td>Symbol table for xxx</td>
</tr>
<tr>
<td>SYMTAB DED..xxx</td>
<td>Symbol table DED for xxx</td>
</tr>
<tr>
<td>USER LABEL..xxx</td>
<td>Source program label for xxx</td>
</tr>
<tr>
<td>xxx</td>
<td>Variable with name xxx. If the variable is not initialized, no text appears against the comment. There is also no static offset if the variable is an array (the static offset can be calculated from the array descriptor, if required).</td>
</tr>
</tbody>
</table>

Variable storage map
For automatic and static internal variables, the variable storage map contains the following information:
- PL/I for MVS & VM identifier name
- Level
- Storage class
- Name of the PL/I for MVS & VM block in which it is declared
- Offset from the start of the storage area, in both decimal and hexadecimal form

If the LIST option is also specified, a map of the static internal and external control sections, called the static storage map, is also produced.

Object code listing
The object code listing consists of the machine instructions and a translation of these instructions into a form that resembles assembler and includes comments, such as source program statement numbers.

The machine instructions are formatted into blocks of code, headed by the statement or line number in the PL/I for MVS & VM source program listing. Generally, only executable statements appear in the listing. DECLARE statements are not normally included. The names of PL/I for MVS & VM variables, rather than the addresses that appear in the machine code, are listed. Special mnemonics are used to refer to some items, including test hooks, descriptors, and address constants.

Statements in the object code listing are ordered by block, as they are sequentially encountered in the source program. Statements in the external procedure are given first, followed by the statements in each inner block. As a result, the order of statements frequently differs from that of the source program.

Every object code listing begins with the name of the external procedure. The actual entry point of the external procedure immediately follows the heading comment REAL ENTRY. The subsequent machine code is the prolog for the block, which performs block activation. The comment PROCEDURE BASE marks the end of
the prolog. Following this is a translation of the first executable statement in the PL/I for MVS & VM source program. Table 45 summarizes the comment used in the listing.

Table 45. Comments in a PL/I for MVS & VM object code listing

<table>
<thead>
<tr>
<th>Comment</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN BLOCK xxx</td>
<td>Indicates the start of the begin block with label xxx</td>
</tr>
<tr>
<td>BEGIN BLOCK NUMBER n</td>
<td>Indicates the start of the begin block with number n</td>
</tr>
<tr>
<td>CALCULATION OF COMMONED EXPRESSION FOLLOWS</td>
<td>Indicates that an expression used more than once in the routine is calculated at this point</td>
</tr>
<tr>
<td>CODE MOVED FROM STATEMENT NUMBER n</td>
<td>Indicates object code moved by the optimization process to a different part of the routine and gives the number of the statement from which it originated</td>
</tr>
<tr>
<td>COMPILER GENERATED SUBROUTINE xxx</td>
<td>Indicates the start of compiler-generated subroutine xxx</td>
</tr>
<tr>
<td>CONTINUATION OF PREVIOUS REGION</td>
<td>Identifies the point at which addressing from the previous routine base recommences</td>
</tr>
<tr>
<td>END BLOCK</td>
<td>Indicates the end of a begin block</td>
</tr>
<tr>
<td>END INTERLANGUAGE PROCEDURE xxx</td>
<td>Identifies the end of an ILC procedure xxx</td>
</tr>
<tr>
<td>END OF COMMON CODE</td>
<td>Identifies the end of code used in running more than one statement</td>
</tr>
<tr>
<td>END OF COMPILER GENERATED SUBROUTINE</td>
<td>Indicates the end of the compiler-generated subroutine</td>
</tr>
<tr>
<td>END PROCEDURE</td>
<td>Indicates the end of a procedure</td>
</tr>
<tr>
<td>END PROGRAM</td>
<td>Indicates the end of the external procedure</td>
</tr>
<tr>
<td>INITIALIZATION CODE FOR xxx</td>
<td>Indicates the start of initialization code for variable xxx</td>
</tr>
<tr>
<td>INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS</td>
<td>Indicates that some of the code that follows was moved from within a loop by the optimization process</td>
</tr>
<tr>
<td>INTERLANGUAGE PROCEDURE xxx</td>
<td>Identifies the start of an implicitly generated ILC procedure xxx</td>
</tr>
<tr>
<td>METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED</td>
<td>Indicates that the order of the code following was changed to optimize the object code</td>
</tr>
<tr>
<td>ON-UNIT BLOCK NUMBER n</td>
<td>Indicates the start of an ON-unit block with number n</td>
</tr>
<tr>
<td>ON-UNIT BLOCK END</td>
<td>Indicates the end of the ON-unit block</td>
</tr>
<tr>
<td>PROCEDURE xxx</td>
<td>Indicates the start of the procedure labeled xxx</td>
</tr>
<tr>
<td>PROCEDURE BASE</td>
<td>Identifies the address loaded into the base register for the procedure</td>
</tr>
<tr>
<td>PROGRAM ADDRESSABILITY REGION BASE</td>
<td>Identifies the address where the routine base is updated if the routine size exceeds 4096 bytes and consequently cannot be addressed from one base</td>
</tr>
<tr>
<td>PROLOGUE BASE</td>
<td>Identifies the start of the prolog code common to all entry points into that procedure</td>
</tr>
<tr>
<td>REAL ENTRY</td>
<td>Precedes the actual executable entry point for a procedure</td>
</tr>
<tr>
<td>STATEMENT LABEL xxx</td>
<td>Identifies the position of source program statement label xxx</td>
</tr>
<tr>
<td>STATEMENT NUMBER n</td>
<td>Identifies the start of code generated for statement number n in the source listing</td>
</tr>
</tbody>
</table>

In certain cases, the compiler uses mnemonics (see Table 46 on page 282) to identify the type of operand in an instruction and, where applicable, follows the mnemonic by the name of a PL/I for MVS & VM variable.
Table 46. PL/I for MVS & VM mnemonics

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A..xxx</td>
<td>Address constant for xxx</td>
</tr>
<tr>
<td>ADD..xxx</td>
<td>Aggregate descriptor for xxx</td>
</tr>
<tr>
<td>BASE..xxx</td>
<td>Base address of variable xxx</td>
</tr>
<tr>
<td>BLOCK..</td>
<td>Identifier created for an otherwise unlabeled block</td>
</tr>
<tr>
<td>CL..n</td>
<td>Compiler-generated label number n</td>
</tr>
<tr>
<td>D..xxx</td>
<td>Descriptor for xxx</td>
</tr>
<tr>
<td>DED..xxx</td>
<td>Data element descriptor for xxx</td>
</tr>
<tr>
<td>HOOK...ENTRY</td>
<td>Debugging tool block entry hook</td>
</tr>
<tr>
<td>HOOK...BLOCK-EXIT</td>
<td>Debugging tool block exit hook</td>
</tr>
<tr>
<td>HOOK...PGM-EXIT</td>
<td>Debugging tool program exit hook</td>
</tr>
<tr>
<td>HOOK...PRE-CALL</td>
<td>Debugging tool pre-call hook</td>
</tr>
<tr>
<td>HOOK...INFO</td>
<td>Additional pre-call hook information</td>
</tr>
<tr>
<td>HOOK...POST-CALL</td>
<td>Debugging tool post call hook</td>
</tr>
<tr>
<td>HOOK...STMT</td>
<td>Debugging tool statement hook</td>
</tr>
<tr>
<td>HOOK...IF-TRUE</td>
<td>Debugging tool IF true hook</td>
</tr>
<tr>
<td>HOOK...IF-FALSE</td>
<td>Debugging tool ELSE hook</td>
</tr>
<tr>
<td>HOOK...WHEN</td>
<td>Debugging tool WHEN true hook</td>
</tr>
<tr>
<td>HOOK...OTHERWISE</td>
<td>Debugging tool OTHERWISE true hook</td>
</tr>
<tr>
<td>HOOK...LABEL</td>
<td>Debugging tool label hook</td>
</tr>
<tr>
<td>HOOK...DO</td>
<td>Debugging tool iterative DO hook</td>
</tr>
<tr>
<td>HOOK...ALLOC</td>
<td>Debugging tool ALLOCATE controlled hook</td>
</tr>
<tr>
<td>WSP..n</td>
<td>Workspace, followed by identifying number n</td>
</tr>
<tr>
<td>L..xxx</td>
<td>Length of variable xxx</td>
</tr>
<tr>
<td>PR..xxx</td>
<td>Pseudoregister vector slot for xxx</td>
</tr>
<tr>
<td>LOCATOR..xxx</td>
<td>Locator for xxx</td>
</tr>
<tr>
<td>RKD..xxx</td>
<td>Record or key descriptor for xxx</td>
</tr>
<tr>
<td>VO..xxx</td>
<td>Virtual origin for xxx (the address where element 0 is held for a one-dimensional array, element 0,0 for a two-dimensional array, and so on)</td>
</tr>
</tbody>
</table>

Generating a Language Environment dump of a PL/I for MVS & VM routine

To generate a dump of a PL/I for MVS & VM routine, you can call either the Language Environment callable service CEE3DMP or PLIDUMP. For information about calling CEE3DMP, see "Generating a Language Environment dump with CEE3DMP" on page 39.

PLIDUMP syntax and options

PLIDUMP calls intermediate PL/I for MVS & VM library routines, which convert most PLIDUMP options to CEE3DMP options. The following list contains PLIDUMP options and the corresponding CEE3DMP option, if applicable.
Some PLIDUMP options do not have corresponding CEE3DMP options, but continue to function as PL/I for MVS & VM default options. The list following the syntax diagram provides a description of those options.

PLIDUMP now conforms to National Language Support standards.

PLIDUMP can supply information across multiple Language Environment enclaves. If an application running in one enclave fetches a main procedure (an action that creates another enclave), PLIDUMP contains information about both procedures.

The syntax and options for PLIDUMP are shown below.

```
Syntax

PLIDUMP (char.-string-exp 1, char.-string-exp 2)

char.-string-exp 1
A dump options character string consisting of one or more of the following values. T, F, C, and A are the default options.
   A All. Results in a dump of all tasks including the ones in the WAIT state.
   B BLOCKS (PL/I for MVS & VM hexadecimal dump). Dumps the control blocks used in Language Environment and member language libraries. For PL/I for MVS & VM, this includes the DSA for every routine on the call chain and PL/I for MVS & VM "global" control blocks, such as Tasking Implementation Appendage (TIA), Task Communication Area (TCA), and the PL/I Tasking Control Block (PTCB). PL/I file control blocks and file buffers are also dumped if the F option is specified.
   C Continue. The routine continues after the dump.
   E Exit. The enclave terminates after the dump. In a multitasking environment, if PLIDUMP is called from the main task, the enclave terminates after the dump. If PLIDUMP is called from a subtask, the subtask and any subsequent tasks created from the subtask terminate after the dump. In a multithreaded environment, if PLIDUMP is called from the Initial Process Thread (IPT), the enclave terminates after the dump. If PLIDUMP is called from a non-IPT, only the non-IPT terminates after the dump.
   F FILE INFORMATION. A set of attributes for all open files is given. The contents of the file buffers are displayed if the B option is specified.
   H STORAGE in hexadecimal. A SNAP dump of the region is produced. A ddname of CEESNAP must be provided to direct the CEESNAP dump report.
   K BLOCKS (when running under CICS). The Transaction Work Area is included.

Note: This option is not supported under Enterprise PL/I.
   NB NOBLOCKS.
```
NF
NOFILES.

NH
NOSTORAGE.

NK
NOBLOCKS (when running under CICS).

NT
NOTRACEBACK.

O
THREAD(CURRENT). Results in a dump of only the current task or current thread (the invoker of PLIDUMP).

S
Stop. The enclave terminates after the dump. In a multitasking environment, regardless of whether PLIDUMP is called from the main task or a subtask, the enclave terminates after the dump. In a multithreaded environment, regardless of whether PLIDUMP is called from the IPT or a non-IPT, the enclave terminates after the dump (in which case there is no fixed order as to which thread terminates first).

T
TRACEBACK. Includes a traceback of all routines on the call chain. The traceback shows transfers of control from either calls or exceptions. BEGIN blocks and ON-units are also control transfers and are included in the trace. The traceback extends backwards to the main program of the current thread.

char.-string-exp 2
A user-identified character string up to 80 characters long that is printed as the dump header.

**PLIDUMP usage notes**

If you use PLIDUMP, the following considerations apply:

- If a routine calls PLIDUMP a number of times, use a unique user-identifier for each PLIDUMP invocation. This simplifies identifying the beginning of each dump.
- In MVS or TSO, you can use ddnames of CEEDUMP, PLIDUMP, or PL1DUMP to direct dump output. If no ddname is specified, CEEDUMP is used.
- The data set defined by the PLIDUMP, PL1DUMP, or CEEDUMP DD statement should specify a logical record length (LRECL) of at least 131 to prevent dump records from wrapping.
- When you specify the H option in a call to PLIDUMP, the PL/I for MVS & VM library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of PLIDUMP 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.
- Support for SNAP dumps using PLIDUMP is provided only under MVS. SNAP dumps are not produced in a CICS environment.
  - If the SNAP does not succeed, the CEE3DMP DUMP file displays the message:
    Snap was unsuccessful
    Failure to define a CEESNAP data set is the most likely cause of an unsuccessful CEESNAP.
  - If the SNAP is successful, CEE3DMP displays the 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.
- To ensure portability across system platforms, use PLIDUMP to generate a dump of your PL/I for MVS & VM routine.
Finding PL/I for MVS & VM information in a dump

The following sections discuss PL/I-specific information located in the following sections of a Language Environment dump:

- Traceback
- Control Blocks for Active Routines
- Control Block Associated with the Thread
- File Status and Attributes

Traceback

Examine the traceback section of the dump, shown in Figure 116, for condition information about your routine and information about the statement number and address where the exception occurred.
**PL/I for MVS & VM task traceback**

A task traceback table is produced for multitasking programs showing the task invocation sequence (trace). For each task, the thread ID, CAA address (identified by TCA address in the dump), event variable address, task variable address, and absolute priority appear in the traceback table. An example is shown in Figure 117.

---

Figure 117. Task traceback section

---

z/OS Language Environment Debugging Guide
If the dump was called from an ON-unit, the type of ON-unit is identified in the traceback as part of the entry information. For ON-units, the values of any relevant condition built-in functions (for example, ONCHAR and ONSOURCE for conversion errors) appear. In cases where the cause of entry into the ON-unit is not stated, usually when the ERROR ON-unit is called, the cause of entry appears in the condition information.

**Statement number and address where error occurred**

This information, which is the point at which the condition that caused entry to the ON-unit occurred, can be found in the traceback section of the dump.

If the condition occurs in compiled code, and you compiled your routine with either GOSTMT or GONUMBER, the statement numbers appear in the dump. To identify the assembler instruction that caused the error, use the traceback information in the dump to find the program unit (PU) offset of the statement number in which the error occurred. Then find that offset and the corresponding instruction in the object code listing.

**Control blocks for active routines**

This section shows the stack frames for all active routines, and the static storage. Use this section of the dump to identify variable values, determine the contents of parameter lists, and locate the timestamp. Figure 119 on page 288 shows this section of the dump.
Control Blocks for Active Routines:

**DSA for CEKKMA**: 20B458B8

*Figure 119. Control blocks for active routines section of the dump (Part 1 of 3)*
Figure 120. Control blocks for active routines section of the dump (Part 2 of 3)
Dynamic save area (LAB1: BEGIN): 20B42430
+000000 20B42430 04255000 20B42430 20B42500 00900E08 0018328 000000B8 20900E80 00900B8E d....................
+000020 20B42450 20902E40 00000001 20B42330 20B42330 000000026 00000008 20B42300 +,1
+000040 20B42470 2090102C 209069C0 000025290 20B42500 20B42500 91A09140 20B42330 00000000 Z...J...J.?
+000060 20B42490 00000000 000000000 000000000 000000000 000000000 000000000 000000000 0002042B303
+000080 20B424B0 20B42530 0000001C 20B424A8 000000000 000000000 000000000 000000000 000000000
+0000A0 20B424D0 2090E0B0 000000004 000000004 000000004 000000004 000000004 000000004 000000004
+0000C0 20B424F0 20B42330 20B423618 00000014 000000000 000000080 000000000 20B42430 20B425C0 800187F8 h..................d
DSA for EXAMPLE: 20B42330
+00000000 20B42330 000000000 000000000 000000000 000000000 000000000 000000000 000000000
+000010 20B42330 000000000 000000000 000000000 000000000 000000000 000000000 000000000
+000020 20B42330 000000000 000000000 000000000 000000000 000000000 000000000 000000000
+000040 20B42330 000000000 000000000 000000000 000000000 000000000 000000000 000000000
+000060 20B42330 000000000 000000000 000000000 000000000 000000000 000000000 000000000
+000080 20B42330 000000000 000000000 000000000 000000000 000000000 000000000 000000000
+0000A0 20B42330 000000000 000000000 000000000 000000000 000000000 000000000 000000000
+0000C0 20B42330 000000000 000000000 000000000 000000000 000000000 000000000 000000000
+0000E0 20B42330 000000000 000000000 000000000 000000000 000000000 000000000 000000000

Figure 121. Control blocks for active routines section of the dump (Part 3 of 3)

Automatic variables
To find automatic variables, use an offset from the stack frame of the block in which they are declared. This information appears in the variable storage map generated when the MAP compiler option is in effect. If you have not used the MAP option, you can determine the offset by studying the listing of compiled code instructions.

Static variables
If your routine is compiled with the MAP option, you can find static variables by using an offset in the variable storage map. If the MAP option is not in effect, you can determine the offset by studying the listing of compiled code.

Based variables
To locate based variables, use the value of the defining pointer. Find this value by using one of the methods described above to find static and automatic variables. If the pointer is itself based, you must find its defining pointer and follow the chain until you find the correct value. The following is an example of typical code for X BASED (P), with P AUTOMATIC:

```
58 60 D 0C8 L 6,P
58 E0 6 000 L 14,X
```

P is held at offset 'X'CS' from register 13. This address points to X.

Take care when examining a based variable to ensure that the pointers are still valid.

Area variables
Area variables are located using one of the methods described above, according to their storage class. The following is an example of typical code: for an area variable A declared AUTOMATIC:

```
41 60 D 0F8 LA 6,A
```

290 z/OS Language Environment Debugging Guide
The area starts at offset XF8' from register 13.

**Variables in areas**
To find variables in areas, locate the area and use the offset to find the variable.

**Contents of parameter lists**
To find the contents of a passed parameter list, first find the register 1 value in the save area of the calling routine's stack frame. Use this value to locate the parameter list in the dump. If R1=0, no parameters passed. For additional information about parameter lists, see the [IBM Enterprise PL/I for z/OS library](www.ibm.com/support/docview.wss?uid=swg27036735).

**Timestamp**
If the TSTAMP compiler installation option is in effect, the date and time of compilation appear within the last 32 bytes of the static internal control section. The last three bytes of the first word give the offset to this information. The offset indicates the end of the timestamp. Register 3 addresses the static internal control section. If the BLOCK option is in effect, the timestamp appears in the static storage section of the dump.

**Control blocks associated with the thread**
This section of the dump, shown in Figure 122 on page 292, includes information about PL/I for MVS & VM fields of the CAA and other control block information.
Figure 122. Control blocks associated with the thread section of the dump (Part 1 of 2)
Figure 123. Control blocks associated with the thread section of the dump (Part 2 of 2)

CAA address

The address of the CAA control block appears in this section of the dump. If the BLOCK option is in effect, the complete CAA (including the PL/I for MVS & VM implementation appendage) appears separately from the body of the dump.

Register 12 addresses the CAA.

File status and attribute information

This part of the dump includes the following information:

- The default and declared attributes of all open files
- Buffer contents of all file buffers
- The contents of FCBs, DCBs, DCLCBs, IOCBs, and control blocks for the process or enclave

PL/I for MVS & VM contents of the Language Environment trace table

Language Environment provides three PL/I for MVS & VM trace table entry types that contain character data:

- Trace entry 100 occurs when a task is created.
- Trace entry 101 occurs when a task that contains the tasking CALL statements is terminated.
• Trace entry 102 occurs when a task that does not contain a tasking CALL statement is terminated.

The format for trace table entries 100, 101, and 102 is shown below. For more information about the Language Environment trace table format, see "Understanding the trace table entry (TTE)" on page 161.

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Name of Calling Task</th>
<th>Name of Called Task</th>
<th>Offset of Call Stmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>Returner R2-R5</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>Returner R12-R14</td>
<td></td>
</tr>
</tbody>
</table>
```

Debugging example of PL/I for MVS & VM routines

This section contains examples of PL/I for MVS & VM routines and instructions for using information in the Language Environment dump to debug them. Important areas in the source code and in the dump for each routine are highlighted.

Subscript range error

Figure 124 illustrates an error caused by an array subscript value outside the declared range. In this example, the declared array value is 10. This routine was compiled with the options LIST, TEST, GOSTMT, and MAP. It was run with the TERMTHDACT(TRACE) option to generate a traceback for the condition.

```
5688-235 IBM PL/I for MVS & VM  Ver 1 Rel 1 Mod 1  27 FEB 07  11:45:18  PAGE 1
OPTIONS SPECIFIED
*PROCESS GOSTMT LIST S STG TEST MAP NOOPTIONS;
5688-235 IBM PL/I for MVS & VM EXAMPLE: PROC OPTIONS(MAIN);
SOURCE LISTING
STMT
  1 EXAMPLE: PROC OPTIONS(MAIN);
  2 DCL Array(10) Fixed bin(31);
  3 DCL (I,Array_End) Fixed bin(31);
  4 On error
     Begin;
  5    On error system;
  6    Call plidump('tbnfs','Plidump called from error On-unit');
     End;
  7    End;
  8   Lab1: Begin;
  9       /* Enable subscriptrange condition */
 10      Array_End = 20;
 11     Do I = 1 to Array_End; /* Loop to initialize array */
 12        Array(I) = 2; /* Set array elements to 2 */
 13    End;
 14   End Example;
```

Figure 124. Example of moving a value outside an array range
Figure 125 shows sections of the dump generated by a call to PLIDUMP.

PLIDUMP Processing started.
PLIDUMP was called from statement number 6 at offset +00000006 from ERR ON-unit with entry address 20900C58

Information for enclave EXAMPLE

Information for thread 8000000000000000

Traceback:

DSA Entry E Offset Statement Load Mod Program Unit Service Status
1 CEEKMAA +0000081C CEEPLPKA CEEKMAA D1908 Call
2 IBMXKOM +000000E2 IBMRE10 IBMXKOM Call
3 ERR ON-unit+00000066 6 EXAMPLE EXAMPLE Call
4 IBMRE1PL +0000065A IBMRE1B IBMRE1PL Call
5 CEEVV010 +0000013A IBMRE10 CEEVV010 Call
6 CEEHDSF +0000017D CEEPLPKA CEEHDSF D1908 Call
7 IBMERRR +0000045A IBMRE1B IBMERRR Exception
8 LABEL: BEGIN+000000E8 11 EXAMPLE EXAMPLE Call
9 EXAMPLE +000000C8 8 EXAMPLE EXAMPLE Call
10 IBMMPMIA +0000051E IBMRE1B IBMMPMIA Call
11 CEEVV010 +00000310 IBMRE10 CEEVV010 Call
12 CEEBBEXT +00000186 CEEPLPKA CEEBBEXT D1908 Call

DSA DSA Addr E Addr PU Addr PU Offset Comp Date Compile Attributes
1 20845888 209F0420 209F0420 +00008538 20061214 CEL
2 00025670 20B1C0A0 20B1C0A0 +000000C2 20061214 CEL
3 20845A88 20900C58 20900E70 +000001BE 20061213 PL/I
4 20845850 00019F50 00019F50 +0000065A 20061213 LIBRARY
5 208457CB 2082F998 2082F998 +0000013A 20061213 LIBRARY
6 208462A8 2088F068 2088F068 +0000170D 20061215 CEL
7 20842500 00011B38 00011B38 +0000045A 20061213 LIBRARY
8 20842430 20900348 20900348 +00000296 20061213 PL/I
9 20842F30 20900376 20900376 +00000000 20061213 PL/I
10 20842178 000201D0 000201D0 +0000051E 20061214 LIBRARY
11 20842208 2082F998 2082F998 +00000310 20061213 LIBRARY
12 20842300 2090D088 2090D088 +00000186 20061215 CEL

Condition Information for Active Routines

Condition Information for IBMERRR (DSA address 20842500)
CIB Address: 20842C7B
Current Condition: IBM0281S A prior condition was promoted to the ERROR condition.
Original Condition:

IBM0421S ONCODE=520 The SUBSCRIPTRANGE condition was raised.
Location:
Program Unit: IBMERRR Entry: IBMERRR Statement: Offset: +0000045A

Storage dump near condition, beginning at location: 0001B772
+000000 0001B772 50500000 50A0CD20 50FA01C 41100008 05E91080 404F4710 A4709104 404F47E0 |&|...B..0.......j. |...j. |...

Control Blocks for Active Routines:

DSA for CEEHDSF: 20842648
+000000 FLAGS.... 0800 member... CEEI BKC..... 20842500 FWC..... 208457CB R14..... A09C083A
+000010 R15..... 00009298 R0.... 00000020 R1..... 209082E8 R2..... 20842C7B 20842330
+000024 R4..... 0093C3C4 R5..... 0000FF7F R6..... 00000001 R7..... 00000007 R8..... A09C0542
+000038 R9..... 20844646 R10..... 208436A7 R11..... A09F0668 R12..... 20909C0 reserved. 0000000 reserved. 208A471C
+00004C NAC..... 208457CB PMAB..... 0000000 reserved. 00000000 reserved. 00000000 MODE..... 00000000 reserved. 00000000
+000064 reserved. 00000000 reserved. 00000000 reserved. 00000000
+000078 reserved. 00000000 reserved. 00000000

DSA for IBMERRR: 20842500
+000000 FLAGS.... 0800 member... 0000 BKC..... 20842500 FWC..... 20842C50 R14..... 8001B784
+000010 R15..... 00A90848 R0.... 00000000 R1..... 20842580 R2..... 00000000 R3..... 20909E08
+000024 R4..... 00025470 R5..... 00002544 R6..... 20842330 R7..... 20842330 R8..... 00000028
+000038 R9..... 00000000 R10..... A0910E0 R11..... 0001B22E R12..... 20909C0 reserved. 00025290
+00004C NAC..... 20842500 PMAB..... 008FF4E reserved. 2090D088 20A9F81C
+000064 reserved. 20909C0 reserved. 20842F0 MODE..... 2082A47 reserved. A0910E0
+000078 reserved. 00000000 reserved. 208A473B

Figure 125. Sections of the Language Environment dump (Part 1 of 2)
To debug this routine, use the following steps:

1. In the dump, PLIDUMP was called by the ERROR ON-unit in statement 6. The traceback information in the dump shows that the exception occurred following statement 11.

2. Locate the Original Condition message in the Condition Information for Active Routines section of the dump. The message is IBM0421S ONCODE=520 The SUBSCRIPT RANGE condition was raised. This message indicates that the exception occurred when an array element value exceeded the subscript range value (in this case, 10). For more information about this message, see e/OS Language Environment Runtime Messages.

3. Locate statement 9 in the routine in Figure 124 on page 294. The instruction is Array_End = 20. This statement assigns a 20 value to the variable Array_End.

4. Statement 10 begins the DO-loop instruction, I = 1 to Array_End. Since the previous instruction (statement 9) specified that Array_End = 20, the loop in statement 10 should run until I reaches a 20 value.
The instruction in statement 2, however, declared a 10 value for the array range. Therefore, when the I value reached 11, the SUBSCRIPTRANGE condition was raised.

The following steps provide another method for finding the value that raised the SUBSCRIPTRANGE condition.

1. Locate the offset of variable I in the variable storage map in Figure 124 on page 294. Use this offset to find the I value at the time of the dump. In this example, the offset is X'C8'.

2. Now, find offset X'C8' from the start of the stack frame for the entry EXAMPLE in Figure 125 on page 295. The block located at this offset contains the value that exceeded the array range, X'B' or 11.

**Calling a nonexistent subroutine**

Figure 127 demonstrates the error of calling a nonexistent subroutine. This routine was compiled with the LIST, MAP, and GOSTMT compiler options. It was run with the TERMTHDACT(DUMP) runtime option to generate a traceback.

---

**Figure 127. Example of calling a nonexistent subroutine**

Figure 128 on page 298 shows the traceback and condition information from the dump.
Figure 128. Sections of the Language Environment dump (Part 1 of 2)
To understand the traceback and debug this example routine, use the following steps:

1. Find the Current Condition message in the Condition Information for Active Routines section of the dump. The message is CEE3201S. The system detected an Operation exception. For more information about this message, see z/OS Language Environment Runtime Messages.

   This section of the dump also provides such information as the name of the active routine and the current statement number at the time of the dump. The location section indicates that the exception occurred at offset X'20900D2C' within entry EXAMPLE1 and that there might have been a bad branch from offset X'+000000C0' statement 7 within entry EXAMPLE1.

2. Locate statement 7 in the routine (Figure 127 on page 297). This statement calls subroutine Prog01. The message CEE3201S, which indicates an operations exception, was generated because of an unresolved external reference.

3. Check the linkage editor output for error messages.

   Divide-by-zero error (Figure 130 on page 300) demonstrates a divide-by-zero error. In this example, the main PL/I for MVS & VM routine passed bad data to a PL/I for MVS & VM subroutine. The bad data in this example is 0, and the error occurred when the...
subroutine SUB1 attempted to use this data as a divisor.

Since variables are not normally displayed in a PLIDUMP dump, this routine included a PUT DATA statement, which generated a listing of arguments and variables used in the routine. Figure 131 shows this output.

Figure 130. PL/I for MVS & VM routine with a divide-by-zero error

Since variables are not normally displayed in a PLIDUMP dump, this routine included a PUT DATA statement, which generated a listing of arguments and variables used in the routine. Figure 131 shows this output.

<table>
<thead>
<tr>
<th>Sample Starting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub1 Starting</td>
</tr>
<tr>
<td>A_NUMBER=</td>
</tr>
<tr>
<td>0 MY_NAME='Tery Gillaspy'</td>
</tr>
<tr>
<td>AN_ARRAY(1)= 1</td>
</tr>
<tr>
<td>AN_ARRAY(2)= 3</td>
</tr>
<tr>
<td>AN_ARRAY(3)= 5</td>
</tr>
</tbody>
</table>

Figure 131. Variables from routine SAMPLE

The routine in Figure 130 was compiled with the LIST compiler option, which generated the object code listing shown in Figure 132 on page 301.
Figure 132. Object code listing from example PL/I for MVS & VM routine

<table>
<thead>
<tr>
<th>STATEMENT NUMBER</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0003A2</td>
<td>58 B0 D 0C8</td>
</tr>
<tr>
<td>0003A6</td>
<td>58 40 B 004</td>
</tr>
<tr>
<td>0003AA</td>
<td>58 90 3 0B4</td>
</tr>
<tr>
<td>0003AE</td>
<td>5C 80 4 004</td>
</tr>
<tr>
<td>0003B2</td>
<td>58 70 3 004</td>
</tr>
<tr>
<td>0003B6</td>
<td>5C 60 4 004</td>
</tr>
<tr>
<td>0003BA</td>
<td>58 80 D 0C0</td>
</tr>
<tr>
<td>0003BE</td>
<td>58 60 B 000</td>
</tr>
<tr>
<td>0003C2</td>
<td>5F 60 4 000</td>
</tr>
<tr>
<td>0003C6</td>
<td>58 E7 6 000</td>
</tr>
<tr>
<td>0003CA</td>
<td>8E 00 0 020</td>
</tr>
<tr>
<td>0003CE</td>
<td>5D E0 8 000</td>
</tr>
<tr>
<td>0003D0</td>
<td>5D F9 6 000</td>
</tr>
</tbody>
</table>

Figure 133 on page 302 shows the Language Environment dump for routine SAMPLE.
Figure 133. Language Environment dump from example PL/I for MVS & VM routine (Part 1 of 3)
Figure 134. Language Environment dump from example PL/I for MVS & VM routine (Part 2 of 3)
To understand the dump information and debug this routine, use the following steps:

1. Notice the title of the dump: PLIDUMP called from error ON-unit. This was the title specified when PLIDUMP was invoked, and it indicates that the ERROR condition was raised and PLIDUMP was called from within the ERROR ON-unit.

2. Locate the messages in the Condition Information section of the dump.

   There are two messages. The current condition message indicates that a prior condition was promoted to the ERROR condition. The promotion of a condition occurs when the original condition is left unhandled (no PL/I for MVS & VM ON-units are assigned to gain control). The original condition message is CEE3209S. The system detected a Fixed Point divide exception. The original condition usually indicates the actual problem. For more information about this message, see z/OS Language Environment Runtime Messages.

3. In the traceback section, note the sequence of calls in the call chain. SAMPLE called SUB1 at statement 11, and SUB1 raised an exception at statement 15, PU offset X'3CE'.

4. Find the statement in the listing for SUB1 that raised the ZERODIVIDE condition. If SUB1 was compiled with GOSTMT and SOURCE, find statement 15 in the source listing.

Since the object listing was generated in this example, you can also locate the actual assembler instruction causing the exception at offset X'3CE' in the object.
listing for this routine, shown in Figure 132 on page 301. Either method shows that divisor was used as the divisor in a divide operation.

5. You can see from the declaration of SUB1 that divisor is a parameter passed from SAMPLE. Because of linkage conventions, you can infer that register 1 in the SAMPLE save area points to a parameter list that was passed to SUB1. divisor is the first parameter in the list.

6. In the SAMPLE DSA, the R1 value is X'20900590'. This is the address of the parameter list, which is located in static storage.

7. Find the parameter list in the stack frame; the address of the first parameter is X'20B42400' and the value of the first parameter is X'00000000'. Thus, the exception occurred when SAMPLE passed a 0 value used as a divisor in subroutine SUB1.
Chapter 8. Debugging Enterprise PL/I routines

This topic contains information that can help you debug applications that contain one or more Enterprise PL/I routines. Following a discussion about potential errors in Enterprise PL/I routines, the first part of this information discusses how to use compiler-generated listings to obtain information about Enterprise PL/I routines, and how to use PLIDUMP to generate a Language Environment dump of an Enterprise PL/I routine. The last part of the chapter provides examples of Enterprise PL/I routines and explains how to debug them using information contained in the traceback information provided in the dump.

Determining the source of errors in Enterprise PL/I routines

Most errors in Enterprise PL/I routines can be identified by the information provided in Enterprise PL/I runtime messages, which begin with the prefix IBM. For a list of these messages, see z/OS Language Environment Runtime Messages.

A malfunction in running an Enterprise PL/I routine can be caused by:
- Logic errors in the source routine
- Invalid use of Enterprise PL/I
- Unforeseen errors
- Invalid input data
- Compiler or runtime routine malfunction
- System malfunction
- Unidentified routine malfunction
- Overlaid storage

Logic errors in the source routine

Errors of this type are often difficult to detect because they often appear as compiler or library malfunctions. Some common errors in source routines are:
- Incorrect conversion from arithmetic data
- Incorrect arithmetic and string manipulation operations
- Unmatched data lists and format lists

Invalid use of Enterprise PL/I

A misunderstanding of the language or a failure to provide the correct environment for using Enterprise PL/I can result in an apparent malfunction of an Enterprise PL/I routine. Any of the following, for example, might cause a malfunction:
- Using uninitialized variables
- Using controlled variables that have not been allocated
- Reading records into incorrect structures
- Misusing array subscripts
- Misusing pointer variables
- Incorrect conversion
- Incorrect arithmetic operations
- Incorrect string manipulation operations

Unforeseen errors

If an error is detected during run time and no ON-unit is provided in the routine to terminate the run or attempt recovery, the job terminates abnormally. However,
the status of a routine at the point where the error occurred can be recorded by using an ERROR ON-unit that contains the following statements. ON ERROR SYSTEM ensures that further errors do not result in a permanent loop.

```
ON ERROR
BEGIN;
  ON ERROR SYSTEM;
  CALL PLIDUMP; /*generates a dump*/
  PUT DATA;   /*displays variables*/
END;
```

**Invalid input data**

A routine should contain checks to ensure that any incorrect input data is detected before it can cause the routine to malfunction. Use the COPY option of the GET statement to check values obtained by stream-oriented input. The values are listed on the file named in the COPY option. If no file name is given, SYSPRINT is assumed.

**Compiler or runtime routine malfunction**

If you are certain that the malfunction is caused by a compiler or runtime routine error, you can either open a PMR or submit an APAR for the error. Meanwhile, you can try an alternative way to perform the operation that is causing the trouble. A bypass is often feasible, since the Enterprise PL/I language frequently provides an alternative method of performing operations.

**System malfunction**

System malfunctions include machine malfunctions and operating system errors. System messages identify these malfunctions and errors to the operator.

**Unidentified routine malfunction**

In most circumstances, an unidentified routine malfunction does not occur when using the compiler. If your routine terminates abnormally without an accompanying Language Environment runtime diagnostic message, the error causing the termination might also be inhibiting the production of a message. Check for the following:

- Your job control statements might be in error, particularly in defining data sets.
- Your routine might overwrite main storage areas containing executable instructions. This can happen if you have accidentally:
  - Assigned a value to a nonexistent array element. For example:
    ```pli
    DCL ARRAY(10);
    ...
    DO I = 1 TO 100;
    ARRAY(I) = VALUE;
    ```

  condition so that each attempt to access an element outside the declared range of subscript values raises the SUBSCRIPTRANGE condition. If there is no ON-unit for this condition, a diagnostic message is printed and the ERROR condition is raised. This facility, though expensive in run time and storage space, is a valuable routine-testing aid.
– Used an incorrect locator value for a locator (pointer or offset) variable. This type of error can occur if a locator value is obtained by means of record-oriented transmission. Ensure that locator values created in one routine, transmitted to a data set, and subsequently retrieved for use in another routine, are valid for use in the second routine.

– Attempted to free a nonbased variable. This can happen when you free a based variable after its qualifying pointer value has been changed. For example:

```pli
DCL A STATIC, B BASED (P);
ALLOCATE B;
P = ADDR(A);
FREE B;
```

– Used the SUBSTR pseudovariable to assign a string to a location beyond the end of the target string. For example:

```pli
DCL X CHAR(3);
I=3
SUBSTR(X,2,I) = 'ABC';
```

To detect this type of error, enable the STRINGRANGE condition during compilation.

**Storage overlay problems**

If you suspect an error in your Enterprise PL/I application is a storage overlay problem, check for the following:

1. The use of a subscript outside the declared bounds (check the SUBSCRIPTRANGE condition)
2. An attempt to assign a string to a target with an insufficient maximum length (check the STRINGSIZE condition)
3. The failure of the arguments to a SUBSTR reference to comply with the rules described for the SUBSTR built-in function (check the STRINGRANGE condition)
4. The loss of significant last high-order (left-most) binary or decimal digits during assignment to an intermediate result or variable or during an input/output operation (check the SIZE condition)
5. The reading of a variable-length file into a variable
6. The misuse of a pointer variable
7. The invocation of a Language Environment callable service with fewer arguments than are required

The first four situations are associated with the listed Enterprise PL/I conditions, all of which are disabled by default. If you suspect one of these problems exists in your routine, use the appropriate condition prefix on the suspected statement or on the BEGIN or PROCEDURE statement of the containing block.

The fifth situation occurs when you read a data record into a variable that is too small. This type of problem only happens with variable-length files. You can often isolate the problem by examining the data in the file information and buffer.
The sixth situation occurs when you misuse a pointer variable. This type of storage overlay is particularly difficult to isolate. There are a number of ways pointer variables can be misused:

- When a READ statement runs with the SET option, a value is placed in a pointer. If you then run a WRITE statement or another READ SET option with another pointer, you overlay your storage if you try to use the original pointer.
- When you try to use a pointer to allocate storage that has already been freed, you can also cause a storage overlay.
- When you attempt to use a pointer set with the ADDR built-in function as a base for data with different attributes, you can cause a storage overlay.

The seventh situation occurs when a Language Environment callable service is passed fewer arguments than its interface requires. The following example might cause a storage overlay because Language Environment assumes that the fourth item in the argument list is the address of a feedback code, when in reality it could be residue data pointing anywhere in storage.

<table>
<thead>
<tr>
<th>Invalid calls</th>
<th>Valid calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL CEEDATE ENTRY OPTIONS(ASM);</td>
<td>DCL CEEDATE ENTRY(<em>,</em>,<em>,</em> OPTIONAL) OPTIONS(ASM);</td>
</tr>
<tr>
<td>CALL CEEDATE(x,y,z); /* invalid */</td>
<td>CALL CEEDATE(x,y,z,<em>); /</em> valid */</td>
</tr>
<tr>
<td>CALL CEEDATE(x,y,z,fc); /* valid */</td>
<td>CALL CEEDATE(x,y,z,fc); /* valid */</td>
</tr>
</tbody>
</table>

### Using Enterprise PL/I compiler listings

The following sections explain how to generate listings that contain information about your routine. Enterprise PL/I listings show machine instructions, constants, and external or internal addresses that the linkage editor resolves. This information can help you find other information, such as variable values, in a dump of an Enterprise PL/I routine.

**Note:** Enterprise PL/I shares a common compiler back-end with C/C++. The Enterprise PL/I assembler listing will, consequently, have a similar form to those from the XL C/C++ compiler.

The compiler listings included below are from the Enterprise PL/I product.

### Generating Enterprise PL/I listings and maps

Table 47 shows compiler-generated listings that you might find helpful when you use information in dumps to debug Enterprise PL/I routines.

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source program</td>
<td>Source program statements</td>
<td>SOURCE</td>
</tr>
<tr>
<td>Cross reference</td>
<td>Cross reference of names with attributes</td>
<td>XREF and ATTRIBUTES</td>
</tr>
<tr>
<td>Aggregate table</td>
<td>Names and layouts of structures and arrays</td>
<td>AGGREGATE</td>
</tr>
<tr>
<td>Variable map</td>
<td>Offsets of automatic and static internal variables (from their defining base)</td>
<td>MAP</td>
</tr>
<tr>
<td>Object code</td>
<td>Contents of the program control section in hexadecimal notation and translated into a pseudo-assembler format.</td>
<td>LIST</td>
</tr>
</tbody>
</table>
### Table 47. Compiler-generated PL/I listings and their contents (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable map, object code, static storage</td>
<td>Same as MAP and LIST options above, plus contents of static internal and static external control sections in hexadecimal notation with comments</td>
<td>MAP and LIST</td>
</tr>
</tbody>
</table>

---

**Finding information in Enterprise PL/I listings**

Figure 136 shows the first two pages of an example Enterprise PL/I routine that was compiled with the LIST, MAP and SOURCE options.

---

5655-H31 IBM(R) Enterprise PL/I for z/OS V3.R6.M0 (Built:20070119)

Options Specified

Install:
Command: s
Line.File Process Statements
1.0 *PROCESS SOURCE LIST MAP;
Install:

5655-H31 IBM(R) Enterprise PL/I for z/OS Compiler Source
Line.File
2.0
3.0 EXAMPLE: PROC OPTIONS(MAIN);
4.0 DCL EXTR ENTRY EXTERNAL;
5.0 DCL A FIXED BIN(31);
6.0 DCL B(2,2) FIXED BIN(31) STATIC EXTERNAL INIT((4)0);
7.0 DCL C CHAR(20) STATIC INIT('SAMPLE CONSTANT');
8.0 DCL D FIXED BIN(31) STATIC;
9.0 DCL E FIXED BIN(31);
10.0 FETCH EXTR;
11.0 CALL EXTR(A,B,C,D,E);
12.0 DISPLAY(C);
13.0 END;

---

**Figure 136. Enterprise PL/I routine compiled with LIST, MAP, and SOURCE**

Figure 137 on page 312 shows the output generated by the LIST and MAP options for this routine, including the pseudo-assembly listing, the external symbol dictionary and reference, the storage offset listing and the static and automatic storage maps. The sections following this example describe the contents of each type of listing.
Figure 137. Compiler-generated listings from example Enterprise PL/I routine (Part 1 of 4)
Figure 138. Compiler-generated listings from example Enterprise PL/I routine (Part 2 of 4)
*** General purpose registers used: 111111110001111
*** Floating point registers used: 1111111000000000
*** Size of register spill area: 512(max) 0(used)
*** Size of dynamic storage: 200
*** Size of executable code: 350
*** CSECT Offset: 72 : 0x48

0001BC 0000 0000

Constant Area

000000 0004CE7E E3D9
5655-H31 IBM(R) Enterprise PL/I for z/OS

OFFSET OBJECT CODE  LINE# FILE# PSEUDO ASSEMBLY LISTING

PPA1: Entry Point Constants

000000 ICCEA166
000004 000001CB
000008 00000000
00000C 00000000
000000 FFE00000
000010 0000000C
000014 00000000
000018 00000000
000019 00000000
00001C 00000000
00001E 00000000
000020 00000000
000024 00000000
000028 00000000
00002C 00000000
000030 00000000
000034 00000000
000038 00000000

PPA1 End

PPA2: Compile Unit Block

000000 0000 3203
000004 FFFF FFO0
000008 0000 0000
00000C FFFF FFO0
000010 0000 0000
000014 0200 0000

PPA2 End

5655-H31 IBM(R) Enterprise PL/I for z/OS

EXTERNAL SYMBOL DICTIONARY

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>ID</th>
<th>ADDR</th>
<th>LENGTH</th>
<th>NAME</th>
<th>TYPE</th>
<th>ID</th>
<th>ADDR</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE1</td>
<td>SD</td>
<td>1</td>
<td>000000</td>
<td>000228</td>
<td>EXAMPLE2</td>
<td>SD</td>
<td>2</td>
<td>000000</td>
<td>00005C</td>
</tr>
<tr>
<td>@EXAMPLE</td>
<td>SD</td>
<td>3</td>
<td>000000</td>
<td>000004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>LD</td>
<td>0</td>
<td>000048</td>
<td>000001</td>
<td>CEESG011</td>
<td>ER</td>
<td>5</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMQFRG</td>
<td>ER</td>
<td>6</td>
<td>000000</td>
<td></td>
<td>IBMQUSB</td>
<td>ER</td>
<td>7</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMQFST</td>
<td>ER</td>
<td>8</td>
<td>000000</td>
<td></td>
<td>CEESTART</td>
<td>ER</td>
<td>9</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>CEEMAIN</td>
<td>SD</td>
<td>10</td>
<td>000000</td>
<td>00000C</td>
<td>IBMINPL</td>
<td>ER</td>
<td>11</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>ER</td>
<td>12</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5655-H31 IBM(R) Enterprise PL/I for z/OS

EXTERNAL SYMBOL CROSS REFERENCE

<table>
<thead>
<tr>
<th>ORIGINAL NAME</th>
<th>EXTERNAL SYMBOL NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE1</td>
<td>EXAMPLE1</td>
</tr>
<tr>
<td>EXAMPLE2</td>
<td>EXAMPLE2</td>
</tr>
<tr>
<td>_EXAMPLE</td>
<td>@EXAMPLE</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>EXAMPLE</td>
</tr>
<tr>
<td>CEESG011</td>
<td>CEESG011</td>
</tr>
<tr>
<td>IBMQFST</td>
<td>IBMQFST</td>
</tr>
<tr>
<td>IBMQFST</td>
<td>IBMQFST</td>
</tr>
<tr>
<td>CEESTART</td>
<td>CEESTART</td>
</tr>
<tr>
<td>CEEMAIN</td>
<td>CEEMAIN</td>
</tr>
<tr>
<td>IBMINPL</td>
<td>IBMINPL</td>
</tr>
</tbody>
</table>

Figure 139. Compiler-generated listings from example Enterprise PL/I routine (Part 3 of 4)
Pseudo assembly listing

The pseudo assembly listing consists of the machine instructions and a translation of these instructions into a form that resembles assembler code. This listing always starts with a small section of non-executable data that records the date and time when the object was produced as well as the version of the compiler used to produce the object. This section ends with a service string which in the listing is followed by the build date for the compiler back-end that generated this part of the listing (and this date may be different from the build date for the compiler front-end that generated the first pages of the listing).

The majority of the pseudo assembly listing consists of the object code arranged in columns that specify for each instruction:
- Its offset.
- the instruction in object code format.
- Its associated line number.
- Its associated file number if non-zero (for example, if from an include file).
• the instruction in mnemonic format.

**External symbol dictionary**
The external symbol dictionary lists all the external symbols generated for this compilation. For each symbol, it also lists its linkage type and size (in hex).

**External symbol cross reference**
The external symbol dictionary cross reference shows for each external symbol the name that will be visible externally to the linker.

**Storage offset listing**
Each line of the storage offset listing contains the following information for each user variable:
• Its name.
• the number of the block in which it was declared.
• the number of the file in which it was declared.
• the number of the line in which it was declared.
• Its class (automatic, static, etc).
• Its location (as appropriate for its class).
• Its byte length in decimal.

This list is sorted by block number and then by name within each block.

**Static map**
Each line of the static storage map contains the following information for each internal static variable:
• Its hexadecimal offset.
• Its byte length in hex.
• Its name.

This list is sorted by the offset of the variables in static. This list of variables may also include compiler-generated variables.

**Automatic map**
Each line of the automatic storage map contains the following information, grouped by named block, for each automatic variable in that block:
• Its hexadecimal offset.
• Its byte length in hex.
• Its name.

These lists are sorted by the offset of the variables in automatic for each block. These lists of variables may also include compiler-generated variables.

---

**Generating a Language Environment dump of an Enterprise PL/I routine**

To generate a dump of an Enterprise PL/I routine, you can call either the Language Environment callable service CEE3DMP or PLIDUMP. For information about calling CEE3DMP, see "Generating a Language Environment dump with CEE3DMP" on page 39.

**PLIDUMP syntax and options**

PLIDUMP calls intermediate Enterprise PL/I library routines, which convert most PLIDUMP options to CEE3DMP options. The following list contains PLIDUMP options and the corresponding CEE3DMP option, if applicable. Some PLIDUMP
options do not have corresponding CEE3DMP options, but continue to function as Enterprise PL/I default options. The list following the syntax diagram provides a description of those options.

PLIDUMP conforms to National Language Support standards. PLIDUMP can supply information across multiple Language Environment enclaves. If an application running in one enclave fetches a main procedure (an action that creates another enclave), PLIDUMP contains information about both procedures. The syntax and options for PLIDUMP are shown below.

**Syntax**

```
PLIDUMP((char.-string-exp 1, char.-string-exp 2))
```

**char.-string-exp 1**

A dump options character string consisting of one or more of the following values. T, F, C, and A are the default options.

- **A** All. Results in a dump of all tasks including the ones in the WAIT state.
- **B** BLOCKS (Enterprise PL/I hexadecimal dump). Dumps the control blocks used in Language Environment and member language libraries. For Enterprise PL/I, this includes the DSA for every routine on the call chain and Enterprise PL/I "global" control blocks, such as Tasking Implementation Appendage (TIA), Task Communication Area (TCA), and the PL/I Tasking Control Block (PTCB). Enterprise PL/I file control blocks and file buffers are also dumped if the F option is specified.
- **C** Continue. The routine continues after the dump.
- **E** Exit. The enclave terminates after the dump. In a multitasking environment, if PLIDUMP is called from the main task, the enclave terminates after the dump. If PLIDUMP is called from a subtask, the subtask and any subsequent tasks created from the subtask terminate after the dump. In a multithreaded environment, if PLIDUMP is called from the Initial Process Thread (IPT), the enclave terminates after the dump. If PLIDUMP is called from a non-IPT, only the non-IPT terminates after the dump.
- **F** FILE INFORMATION. A set of attributes for all open files is given. The contents of the file buffers are displayed if the B option is specified.
- **H** STORAGE in hexadecimal. A SNAP dump of the region is produced. A ddname of CEESNAP must be provided to direct the CEESNAP dump report.
- **K** BLOCKS (when running under CICS). The Transaction Work Area is included.

**Note:** This option is not supported under Enterprise PL/I.

- **NB** NOBLOCKS.
- **NF** NOFILES.
- **NH** NOSTORAGE.
NK NOBLOCKS (when running under CICS).

NT NOTRACEBACK.

O THREAD(CURRENT). Results in a dump of only the current task or current thread (the invoker of PLIDUMP).

S Stop. The enclave terminates after the dump. In a multitasking environment, regardless of whether PLIDUMP is called from the main task or a subtask, the enclave terminates after the dump. In a multithreaded environment, regardless of whether PLIDUMP is called from the IPT or a non-IPT, the enclave terminates after the dump (in which case there is no fixed order as to which thread terminates first).

T TRACEBACK. Includes a traceback of all routines on the call chain. The traceback shows transfers of control from either calls or exceptions. BEGIN blocks and ON-units are also control transfers and are included in the trace. The traceback extends backwards to the main program of the current thread.

char.-string-exp 2
A user-identified character string up to 80 characters long that is printed as the dump header.

PLIDUMP usage notes

If you use PLIDUMP, the following considerations apply:

- If a routine calls PLIDUMP a number of times, use a unique user-identifier for each PLIDUMP invocation. This simplifies identifying the beginning of each dump.
- In MVS or TSO, you can use ddnames of CEEDUMP, PLIDUMP, or PL1DUMP to direct dump output. If no ddname is specified, CEEDUMP is used.
- The data set defined by the PLIDUMP, PL1DUMP, or CEEDUMP DD statement should specify a logical record length (LRECL) of at least 131 to prevent dump records from wrapping.
- When you specify the H option in a call to PLIDUMP, the Enterprise PL/I library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of PLIDUMP 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.
- Support for SNAP dumps using PLIDUMP is provided only under MVS. SNAP dumps are not produced in a CICS environment.
  - If the SNAP does not succeed, the CEE3DMP DUMP file displays the message:
    ```
    Snap was unsuccessful
    Failure to define a CEESNAP data set is the most likely cause of an unsuccessful CEESNAP.
    ```
  - If the SNAP is successful, CEE3DMP displays the message, where nnn corresponds to the SNAP identifier described above. An unsuccessful SNAP does not result in an incrementation of the identifier.
    ```
    Snap was successful; snap ID = nnn
    ```
- To ensure portability across system platforms, use PLIDUMP to generate a dump of your Enterprise PL/I routine.
Finding Enterprise PL/I information in a dump

The following sections discuss Enterprise PL/I-specific information located in the following sections of a Language Environment dump:

- Traceback
- Control Blocks for Active Routines
- Control Block Associated with the Thread
- File Status and Attributes

Traceback

Examine the traceback section of the dump, shown in Figure 141 on page 320, for condition information about your routine and information about the statement number and address where the exception occurred.
Figure 141. Traceback section of dump (Enterprise PL/I)

Condition information

If the dump was called from an ON-unit, the type of ON-unit is identified in the traceback as part of the entry information. For ON-units, the values of any relevant condition built-in functions (for example, ONCHAR and ONSOURCE for conversion errors) appear. In cases where the cause of entry into the ON-unit is not stated, usually when the ERROR ON-unit is called, the cause of entry appears in the condition information.
Statement number and address where error occurred
This information, which is the point at which the condition that caused entry to
the ON-unit occurred, can be found in the traceback section of the dump. If the
condition occurs in compiled code, and you compiled your routine with either
GOSTMT or GONUMBER, the statement numbers appear in the dump. To identify
the assembler instruction that caused the error, use the traceback information in the
dump to find the program unit (PU) offset of the statement number in which the
error occurred. Then find that offset and the corresponding instruction in the object
code listing.

Control blocks for active routines
This section shows the stack frames for all active routines, and the static storage.
Use this section of the dump to identify variable values, determine the contents of
parameter lists, and locate the timestamp. Figure 142 on page 322 shows this
section of the dump.
Automatic variables

To find automatic variables, use an offset from the stack frame of the block in which they are declared. This information appears in the variable storage map generated when the MAP compiler option is in effect. If you have not used the MAP option, you can determine the offset by studying the listing of compiled code instructions.

Static variables

If your routine is compiled with the MAP option, you can find static variables by using an offset in the variable storage map. If the MAP option is not in effect, you can determine the offset by studying the listing of compiled code.
**Based variables**

To locate based variables, use the value of the defining pointer. Find this value by using one of the methods described above to find static and automatic variables. If the pointer is itself based, you must find its defining pointer and follow the chain until you find the correct value.

The following is an example of typical code for X BASED (P), with P AUTOMATIC. P is held at offset X'C8' from register 13. This address points to X.

```
58 60 D 0CB  L 6,P
58 60 6 000  L 14,X
```

Take care when examining a based variable to ensure that the pointers are still valid.

**Area variables**

Area variables are located using one of the methods described above, according to their storage class.

The following is an example of typical code: for an area variable A declared AUTOMATIC. The area starts at offset X'F8' from register 13.

```
41 60 D 0F8  LA 6,A
```

**Variables in areas**

To find variables in areas, locate the area and use the offset to find the variable.

**Contents of parameter lists**

To find the contents of a passed parameter list, first find the register 1 value in the save area of the calling routine's stack frame. Use this value to locate the parameter list in the dump. If R1=0, no parameters passed.

**Control blocks associated with the thread**

This section of the dump, shown in Figure 143 on page 324, includes information about Enterprise PL/I fields of the CAA and other control block information.
Figure 143. Control blocks associated with the thread section of the dump (Enterprise PL/I) (Part 1 of 2)
Figure 144. Control blocks associated with the thread section of the dump (Enterprise PL/I) (Part 2 of 2)

CAA address
The address of the CAA control block appears in this section of the dump. If the BLOCK option is in effect, the complete CAA (including the Enterprise PL/I implementation appendage) appears separately from the body of the dump. Register 12 addresses the CAA.

File status and attribute information
This part of the dump includes the following information:
- The default and declared attributes of all open files
- Buffer contents of all file buffers
- The contents of FCBs, DCBs, DCLCBs, IOCBs, and control blocks for the process or enclave
Enterprise PL/I contents of the Language Environment trace table

Language Environment provides three Enterprise PL/I trace table entry types that contain character data:

- Trace entry 100 occurs when a task is created.
- Trace entry 101 occurs when a task that contains the tasking CALL statements is terminated.
- Trace entry 102 occurs when a task that does not contain a tasking CALL statement is terminated.

The format for trace table entries 100, 101, and 102 follows. For more information about the Language Environment trace table format, see “Understanding the trace table entry (TTE)” on page 161.

```plaintext
–> (100) NameOfCallingTask NameOfCalledTask OffsetOfCallStmt
   UserAgrPtr CalledTaskPtr TaskVarPtr EventVarPtr
   PriorityPtr CallingR2-R5 CallingR12-R14
–> (101) NameOfReturnTask ReturnerR2-R5 ReturnerR12-R14
–> (102) NameOfReturnTask
```

Debugging example of Enterprise PL/I routines

This section contains examples of Enterprise PL/I routines and instructions for using information in the Language Environment dump to debug them. Important areas in the source code and in the dump for each routine are highlighted.

Subscript range error

Figure 145 on page 327 illustrates an error caused by an array subscript value outside the declared range. In this example, the declared array value is 10. This routine was compiled with the options LIST, TEST, GONUMBER, and MAP. It was run with the TERMTHDACT(TRACE) option to generate a traceback for the condition.
Figure 145. Example of moving a value outside an array range (Enterprise PL/I)

Figure 146 on page 328 shows sections of the dump generated by a call to PLIDUMP.
Information for enclave EXAMPLE

Information for thread 8000000000000000

Traceback:

<table>
<thead>
<tr>
<th>DSA</th>
<th>Addr E Addr</th>
<th>Addr</th>
<th>PU Addr</th>
<th>PU Offset</th>
<th>Comp Date</th>
<th>Compile Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11A3D50</td>
<td>11A4A38</td>
<td>11A4AE3B</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>2</td>
<td>11A3D70</td>
<td>11B02040</td>
<td>11B020000</td>
<td>+00000014</td>
<td>20070131</td>
<td>ENTRIES Lib/EBCDIC HFP</td>
</tr>
<tr>
<td>3</td>
<td>11A3D80</td>
<td>11A7A9B8</td>
<td>11A7A9B8</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>4</td>
<td>11A3D90</td>
<td>11A3D900</td>
<td>11A3D900</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>5</td>
<td>11A3D97B</td>
<td>11A4D62B</td>
<td>11A4D62B</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>6</td>
<td>11A3A0B8</td>
<td>11A4C58B</td>
<td>11A4C58B</td>
<td>+00000014</td>
<td>20061214</td>
<td>CEL</td>
</tr>
<tr>
<td>7</td>
<td>11A3A0B8</td>
<td>11A4A4A8</td>
<td>11A4A4A8</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>8</td>
<td>11A3A41B</td>
<td>11A4A41B</td>
<td>11A4A41B</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>9</td>
<td>11A3A41B</td>
<td>11A4A41B</td>
<td>11A4A41B</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>10</td>
<td>11A3A41B</td>
<td>11A4A41B</td>
<td>11A4A41B</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>11</td>
<td>11A3A41B</td>
<td>11A4A41B</td>
<td>11A4A41B</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>12</td>
<td>11A3A41B</td>
<td>11A4A41B</td>
<td>11A4A41B</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>13</td>
<td>11A3A41B</td>
<td>11A4A41B</td>
<td>11A4A41B</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>14</td>
<td>11A3A41B</td>
<td>11A4A41B</td>
<td>11A4A41B</td>
<td>+00000014</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Condition Information for (DSA address 11A3A68B)

CIB Address: 11A3B17B
Current Condition: IBM02615
A prior condition was promoted to the ERROR condition.

Original Condition: IBM04215 ONCODE=520
The SUBSCRIPTRANGE condition was raised.

Location:

Program Unit: Entry: IBMERRI Statement: Offset: +000000AA

Storage dump near condition, beginning at location: 11A4AAB2
+000000 11A4AAB2 4100D998 5000D998 5040D998 5040D990 8201F000 43002016 A7010080 A7840009 |...&.&.&&...&.&...&...&...&...&...

Figure 146. Sections of the Language Environment dump (Part 1 of 2)
Figure 147. Sections of the Language Environment dump (Part 2 of 2)

To debug this routine, use the following steps:

1. In the dump, PLIDUMP was called by the ERROR ON-unit in statement 9. The traceback information in the dump shows that the exception occurred following statement 16.
Note: In the Language Environment dumps, the columns and messages refer to "statements", but the numbers are actually (for Enterprise PL/I) the line numbers from the source file.

2. Locate the Original Condition message in the Condition Information for Active Routines section of the dump. The message is IBM0421S ONCODE=520 The SUBSCRIPTRANGE condition was raised. This message indicates that the exception occurred when an array element value exceeded the subscript range value (in this case, 10). For more information about this message, see z/OS Language Environment Runtime Messages.

3. Locate statement 14 in the routine in Figure 145 on page 327. The instruction is Array_End = 20. This statement assigns a 20 value to the variable Array_End.

4. Statement 15 begins the DO-loop instruction Do I = 1 to Array_End. Since the previous instruction (statement 14) specified that Array_End = 20, the loop in statement 10 should run until I reaches a 20 value.

   The instruction in statement 4, however, declared a 10 value for the array range. Therefore, when the I value reached 11, the SUBSCRIPTRANGE condition was raised.

The following steps provide another method for finding the value that raised the SUBSCRIPTRANGE condition.

1. Locate the offset of variable I in the storage offset listing in Figure 145 on page 327. Use this offset to find the I value at the time of the dump. In this example, the offset is X'E8'.

2. Now find offset X'E8' from the start of the stack frame for the entry EXAMPLE in Figure 146 on page 328. The block located at this offset contains the value that exceeded the array range, XB' or 11.

**Calling a nonexistent subroutine**

Figure 148 on page 331 demonstrates the error of calling a nonexistent subroutine. This routine was compiled with the LIST, MAP, and GONUMBER compiler options. It was run with the TERMTHDACT(DUMP) runtime option to generate a traceback.
Figure 148. Example of calling a nonexistent subroutine (Enterprise PL/I)

Figure 149 on page 332 shows the traceback and condition information sections from the dump.
Figure 149. Traceback and condition information of the Language Environment dump (Enterprise PL/I) (Part 1 of 2)
To understand the traceback and debug this example routine, use the following steps:

1. Find the Current Condition message in the Condition Information for Active Routines section of the dump. The message is CEE3201S The system detected an Operation exception. For more information about this message, see z/OS Language Environment Runtime Messages.

This section of the dump also provides such information as the name of the active routine and the current statement number at the time of the dump. The Location section indicates that the exception occurred at offset X’089009AB’ within entry EXAMPLE1 and that there may have been a bad branch from offset X’+00001AE’ statement 12 within entry EXAMPLE1.

2. Locate statement 12 in the routine (Figure 148 on page 331). This statement calls subroutine Prog01. The message CEE3201S, which indicates an operations exception, was generated because of an unresolved external reference.
3. Check the linkage editor output for error messages.

**Divide-by-zero error**

Figure 151 demonstrates a divide-by-zero error. In this example, the main Enterprise PL/I routine passed bad data to an Enterprise PL/I subroutine. The bad data in this example is 0, and the error occurred when the subroutine SUB1 attempted to use this data as a divisor.

![Figure 151](image)

Because variables are not usually displayed in a PLIDUMP dump, this routine included a PUT DATA statement, which generated a listing of arguments and variables used in the routine. Figure 152 shows this output.

![Figure 152](image)

The routine in Figure 151 was compiled with the LIST compiler option, which generated the object code listing shown in Figure 153 on page 335.
Figure 153. Object code listing from example Enterprise PL/I routine

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>OBJECT CODE</th>
<th>LINE#</th>
<th>FILE#</th>
<th>PSEUDO ASSEMBLY LISTING</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td></td>
<td>000021</td>
<td></td>
<td>SUB1 DS 00</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td>000021</td>
<td></td>
<td>B 34,(r15)</td>
</tr>
<tr>
<td>000004</td>
<td>01C3C5C</td>
<td></td>
<td></td>
<td>CEE eyecatcher</td>
</tr>
<tr>
<td>000008</td>
<td>00000160</td>
<td></td>
<td></td>
<td>DSA size</td>
</tr>
<tr>
<td>00000C</td>
<td></td>
<td>00000848</td>
<td></td>
<td>-A(PPL1-SUB1)</td>
</tr>
<tr>
<td>000136</td>
<td>5820 014C</td>
<td>000027</td>
<td></td>
<td>L r2,#SR_PARM(3(r13,332)</td>
</tr>
<tr>
<td>00013A</td>
<td>5820 2008</td>
<td>000027</td>
<td></td>
<td>L r2,addrARRAY1(r2,0)</td>
</tr>
<tr>
<td>00013E</td>
<td>5840 2000</td>
<td>000027</td>
<td></td>
<td>L r4_shadow1(r2,0)</td>
</tr>
<tr>
<td>000142</td>
<td>5820 014C</td>
<td>000027</td>
<td></td>
<td>L r2,#SR_PARM(3(r13,332)</td>
</tr>
<tr>
<td>000146</td>
<td>5820 2008</td>
<td>000027</td>
<td></td>
<td>L r2,addrARRAY1(r2,0)</td>
</tr>
<tr>
<td>00014A</td>
<td>5820 2004</td>
<td>000027</td>
<td></td>
<td>L r2_shadow1(r2,4)</td>
</tr>
<tr>
<td>00014E</td>
<td>5040 0150</td>
<td>000027</td>
<td></td>
<td>ST r4,wttemp1(r13,336)</td>
</tr>
<tr>
<td>000152</td>
<td>5020 0144</td>
<td>000027</td>
<td></td>
<td>ST r2_temp15(r13,324)</td>
</tr>
<tr>
<td>000156</td>
<td>5820 0144</td>
<td>000027</td>
<td></td>
<td>L r2_temp15(r13,324)</td>
</tr>
<tr>
<td>00015A</td>
<td>5850 2004</td>
<td>000027</td>
<td></td>
<td>L r5_shadow2(r2,4)</td>
</tr>
<tr>
<td>00015E</td>
<td>5840 0150</td>
<td>000027</td>
<td></td>
<td>L r4,wttemp1(r13,336)</td>
</tr>
<tr>
<td>000162</td>
<td>5820 0144</td>
<td>000027</td>
<td></td>
<td>L r2_temp15(r13,324)</td>
</tr>
<tr>
<td>000166</td>
<td>5820 2000</td>
<td>000027</td>
<td></td>
<td>L r2_shadow2(r2,0)</td>
</tr>
<tr>
<td>00016A</td>
<td>1382</td>
<td>000027</td>
<td></td>
<td>LCR r8,r2</td>
</tr>
<tr>
<td>00016C</td>
<td>5820 014C</td>
<td>000027</td>
<td></td>
<td>L r2,#SR_PARM(3(r13,332)</td>
</tr>
<tr>
<td>000170</td>
<td>5820 2008</td>
<td>000027</td>
<td></td>
<td>L r2,addrARRAY1(r2,0)</td>
</tr>
<tr>
<td>000174</td>
<td>5890 2000</td>
<td>000027</td>
<td></td>
<td>L r9_shadow1(r2,0)</td>
</tr>
<tr>
<td>000178</td>
<td>5820 014C</td>
<td>000027</td>
<td></td>
<td>L r2,#SR_PARM(3(r13,332)</td>
</tr>
<tr>
<td>00017C</td>
<td>5820 2008</td>
<td>000027</td>
<td></td>
<td>L r2,addrARRAY1(r2,0)</td>
</tr>
<tr>
<td>000180</td>
<td>5820 2004</td>
<td>000027</td>
<td></td>
<td>L r2_shadow1(r2,4)</td>
</tr>
<tr>
<td>000184</td>
<td>5090 0154</td>
<td>000027</td>
<td></td>
<td>ST r3,wttemp2(r13,340)</td>
</tr>
<tr>
<td>000188</td>
<td>1E58</td>
<td>000027</td>
<td></td>
<td>ALR r5,r8</td>
</tr>
<tr>
<td>00018A</td>
<td>4145 4000</td>
<td>000027</td>
<td></td>
<td>LA r4,Address(r5,r4,0)</td>
</tr>
<tr>
<td>00018E</td>
<td>5040 0158</td>
<td>000027</td>
<td></td>
<td>ST r4,wttemp3(r13,344)</td>
</tr>
<tr>
<td>000192</td>
<td>5020 0140</td>
<td>000027</td>
<td></td>
<td>ST r2_temp14(r13,320)</td>
</tr>
<tr>
<td>000196</td>
<td>5820 0140</td>
<td>000027</td>
<td></td>
<td>L r2_temp14(r13,320)</td>
</tr>
<tr>
<td>00019A</td>
<td>5840 2004</td>
<td>000027</td>
<td></td>
<td>L r4_shadow2(r2,4)</td>
</tr>
<tr>
<td>00019E</td>
<td>8940 0001</td>
<td>000027</td>
<td></td>
<td>SLL r4,1</td>
</tr>
<tr>
<td>0001A2</td>
<td>5820 0154</td>
<td>000027</td>
<td></td>
<td>L r2,wttemp2(r13,340)</td>
</tr>
<tr>
<td>0001A6</td>
<td>5850 0140</td>
<td>000027</td>
<td></td>
<td>L r5_temp14(r13,320)</td>
</tr>
<tr>
<td>0001A9</td>
<td>5850 5000</td>
<td>000027</td>
<td></td>
<td>L r5_shadow2(r5,0)</td>
</tr>
<tr>
<td>0001A8</td>
<td>1355</td>
<td>000027</td>
<td></td>
<td>LCR r5,r5</td>
</tr>
<tr>
<td>0001B0</td>
<td>1E45</td>
<td>000027</td>
<td></td>
<td>ALR r4,r5</td>
</tr>
<tr>
<td>0001B2</td>
<td>5884 2000</td>
<td>000027</td>
<td></td>
<td>L r8_shadow2(r4,r2,0)</td>
</tr>
<tr>
<td>0001B6</td>
<td>5820 014C</td>
<td>000027</td>
<td></td>
<td>L r2,#SR_PARM(3(r13,332)</td>
</tr>
<tr>
<td>0001BA</td>
<td>5820 2000</td>
<td>000027</td>
<td></td>
<td>L r2,addrDIVISOR(r2,0)</td>
</tr>
<tr>
<td>0001BE</td>
<td>5820 2000</td>
<td>000027</td>
<td></td>
<td>L r2_shadow2(r2,0)</td>
</tr>
<tr>
<td>0001C2</td>
<td>8EB0 0020</td>
<td>000027</td>
<td></td>
<td>SRDA r8,32</td>
</tr>
<tr>
<td>0001C6</td>
<td>10B2</td>
<td>000027</td>
<td></td>
<td>DRR r8,r2</td>
</tr>
<tr>
<td>0001C8</td>
<td>1E49</td>
<td>000027</td>
<td></td>
<td>LR r4,r9</td>
</tr>
<tr>
<td>0001CA</td>
<td>5820 0158</td>
<td>000027</td>
<td></td>
<td>L r2,wttemp3(r13,344)</td>
</tr>
<tr>
<td>0001CE</td>
<td>5040 2000</td>
<td>000027</td>
<td></td>
<td>ST r4_shadow2(r2,0)</td>
</tr>
</tbody>
</table>

Figure 154 on page 336 shows the Language Environment dump for routine SAMPLE.
**Figure 154. Language Environment dump from example Enterprise PL/I routine (Part 1 of 2)**
To understand the dump information and debug this routine, use the following steps:

1. Notice the title of the dump: PLIDUMP called from error ON-unit. This was the title specified when PLIDUMP was invoked, and it indicates that the ERROR condition was raised and PLIDUMP was called from within the ERROR ON-unit.

2. Locate the messages in the Condition Information section of the dump.
   There are two messages. The current condition message indicates that a prior condition was promoted to the ERROR condition. The promotion of a condition occurs when the original condition is left unhandled (no Enterprise PL/I ON-units are assigned to gain control). The original condition message is CEE3209S The system detected a fixed-point divide exception. The original condition usually indicates the actual problem. For more information about this message, see z/OS Language Environment Runtime Messages.

3. In the traceback section, note the sequence of calls in the call chain. SAMPLE called SUB1 at statement 19, and SUB1 raised an exception at statement 27, PU offset X'1C6'.

4. Find the statement in the listing for SUB1 that raised the ZERODIVIDE condition. If SUB1 was compiled with GOSTMT and SOURCE, find statement 27 in the source listing.
   Since the object listing was generated in this example, you can also locate the actual assembler instruction causing the exception at offset X'1C6' in the object listing for this routine, shown in Figure 153 on page 335. Either method shows that divisor was loaded into register 2 (r2) and used as the divisor in a divide operation.

5. You can see from the declaration of SUB1 that divisor is a parameter passed from SAMPLE. Because of linkage conventions, you can infer that register 1 in the SAMPLE save area points to a parameter list that was passed to SUB1. divisor is the first parameter in the list.

6. In the SAMPLE DSA, the R1 value is X'11A3B450'. This is the address of the parameter list, which is located in static storage.

7. Find the parameter list in the stack frame; the address of the first parameter is X'11A3B484' and the value of the first parameter is X'00000000'. Thus, the exception occurred when SAMPLE passed a 0 value used as a divisor in subroutine SUB1.
Chapter 9. Debugging under CICS

This section provides information for debugging under the Customer Information Control System (CICS). The following sections explain how to access debugging information under CICS, and describe features unique to debugging under CICS.

Use the following list as a quick reference for debugging information:

- Language Environment runtime messages (CESE transient data queue)
- Language Environment traceback (CESE transient data queue)
- Language Environment dump output (CESE transient data queue)
- CICS Transaction Dump (CICS DFHDMPA or DFHDMPB data set)
- Language Environment abend and reason codes (system console)
- Language Environment return codes to CICS (system console)

If the EXEC CICS HANDLE ABEND command is active and the application, or CICS, initiates an abend or application interrupt, then Language Environment does not produce any runtime messages, tracebacks, or dumps.

If EXEC CICS ABEND NODUMP is issued, then no Language Environment dumps or CICS transaction dumps are produced.

Accessing debugging information

The following sections list the debugging information available to CICS users, and describe where you can find this information.

Under CICS, the Language Environment runtime messages, Language Environment traceback, and Language Environment dump output are written to the CESE transient data queue. The transaction identifier, terminal identifier, date, and time precede the data in the queue. For detailed information about the format of records written to the transient data queue, see z/OS Language Environment Programming Guide.

The CESE transient data queue is defined in the CICS destination control table (DCT). The CICS macro DFHDCCT is used to define entries in the DCT. See CICS Resource Definition Guide for a detailed explanation of how to define a transient data queue in the DCT. If you are not sure how to define the CESE transient data queue, see your system programmer.

Locating Language Environment runtime messages

Under CICS, Language Environment runtime messages are written to the CESE transient data queue. The following example shows a Language Environment message that appears when an application abends due to an unhandled condition from an EXEC CICS command.

```
P039UTV9 199109161453313 CEE3250C The System or User ABEND AEIO was issued.
P039UTV9 199109161453313 From program unit UT9CVERI at entry point UT9CVERIT +0000001E at P039UTV9 199109161453313 at offset address 0006051E.
```
Locating the Language Environment traceback

Under CICS, the Language Environment traceback is written to the CESE transient data queue. Because Language Environment invokes your application routine, the Language Environment routines that invoked your routine appear in the traceback. Figure 156 shows an example Language Environment traceback written to the CESE transient data queue. Data unnecessary for this example has been replaced by ellipses.

---

Locating the Language Environment dump

Under CICS, the Language Environment dump output is written to the CESE transient data queue. For active routines, the Language Environment dump contains the traceback, condition information, variables, storage, and control block
information for the thread, enclave, and process levels. Use the Language Environment dump with the CICS transaction dump to locate problems when operating under CICS. For a sample Language Environment dump, see "Understanding the Language Environment dump" on page 47.

Using CICS transaction dump
The CICS transaction dump is generated to the DFHDMPA or DFHDMPB data set. The offline CICS dump utility routine converts the transaction dump into formatted, understandable output.

The CICS transaction dump contains information for the storage areas and resources associated with the current transaction. This information includes the Communication Area (COMMAREA), Transaction Work Area (TWA), Exec Interface Block (EIB), and any storage obtained by the CICS EXEC commands. This information does not appear in the Language Environment dump. It can be helpful to use the CICS transaction dump with the Language Environment dump to locate problems when operating under CICS.

When the location of an error is uncertain, it can be helpful to insert EXEC CICS DUMP statements in and around the code suspected of causing the problem. This generates CICS transaction dumps close to the error for debugging reference.

For information about interpreting CICS dumps, see CICS Problem Determination Guide.

Using CICS register and program status word contents
When a routine interrupt occurs (code = ASRA) and a CICS dump is generated, CICS formats the contents of the program status word (PSW) and the registers at the time of the interrupt. This information is also contained in the CICS trace table entry marked SSRP * EXEC* – ABEND DETECTED. For the format of the information contained in this trace entry, see CICS Data Areas, KERRD - KERNEL ERROR DATA.

The address of the interrupt can be found from the second word of the PSW, giving the address of the instruction following the point of interrupt. The address of the entry point of the function can be subtracted from this address. The offset compared to this listing gives the statement that causes the interrupt.

For C routines, you can find the address of the entry point in register 3.

If register 15 is corrupted, the contents of the first load module of the active enclave appear in the program storage section of the CICS transaction dump.

Using Language Environment abend and reason codes
An application can end with an abend in two ways:
- User-specified abend (that is, an abend requested by the assembler user exit or the ABTERMENC runtime option).
- Language Environment-detected unrecoverable error (in which case there is no Language Environment condition handling).

When Language Environment detects an unrecoverable error under CICS, Language Environment terminates the transaction with an EXEC CICS abend. The abend code has a number between 4000 and 4095. A write-to-operator (WTO) is performed to write a CEE1000S message to the system console. This message
contains the abend code and its associated reason code. The WTO is performed only for unrecoverable errors detected by Language Environment. No WTO occurs for user-requested abends.

Although this type of abend is performed only for unrecoverable error conditions, an abend code of 4000–4095 does not necessarily indicate an internal error within Language Environment. For example, an application routine can write a variable outside its storage and corrupt the Language Environment control blocks.

Possible causes of a 4000–4095 abend are corrupted Language Environment control blocks and internal Language Environment errors. For more information about abend codes 4000–4095, see "z/OS Language Environment Runtime Messages".

Following is a sample Language Environment abend and reason code. Abend codes appear in decimal, and reason codes appear in hexadecimal.

```
12.34.27 JOB05585 IEF4501 XCEPI1103 GO CEPI1103 - ABEND=5000 U4094 REASON=0000002C
```

### Using Language Environment return codes to CICS

When the Language Environment condition handler encounters a severe condition that is specific to CICS, the condition handler generates a CICS-specific return code. This return code is written to the system console. Possible causes of a Language Environment return code to CICS are:

- Incorrect region size
- Incorrect DCT
- Incorrect CSD definitions

For a list of the reason codes written only to CICS, see "z/OS Language Environment Runtime Messages". The following example shows a sample of a return code that was returned to CICS.

```
+DFHAP1200I
LE03CC01 A CICS request to Language Environment has failed. Reason code '0012030'.
```

### Activating Language Environment feature trace records under CICS

Activating Language Environment feature trace records under CICS will allow users to monitor and determine the activity of a transaction. By activating the feature trace records, Level 2 trace points are added inside Language Environment at these significant points:

- Event Handle
- Set anchor
- Gives R13 and parameters before call

These trace points are useful for any support personnel that needs to know what happened inside Language Environment from a CICS call.

The function will be enabled by the existing CICS transactions. A user must enable the AP domain level 2 in order to include the Language Environment trace points. For more information on activating the CICS trace, see "CICS Diagnosis Reference".

Every time CICS calls Language Environment, the feature trace is activated under the Extended Runtime Library Interface (ERTLI). The trace can be seen in CICS
transaction dumps. Feature trace entries are formatted in a similar way to CICS trace items. There are three formats: ABBREV, SHORT & FULL. The ABBREV version (Figure 157) just formats the heading line for each trace point and is laid out in a similar way to CICS trace entries.

The Domain Name field is replaced with a "Feature" short name (for example, Lang.Env.) and module name (for example, CEE.....) which are coded into the "Feature Trace" initialization (short name) and header formatting call (module name). See the following macro example.

The FULL version includes the heading from the ABBREV version and then dumps each captured block in Hex and Character formats. For an example, see Figure 158 on page 344.

---

**Figure 157. CICS trace output in the ABBREV format.**

The Domain Name field is replaced with a "Feature" short name (for example, Lang.Env.) and module name (for example, CEE.....) which are coded into the "Feature Trace" initialization (short name) and header formatting call (module name). See the following macro example.

The FULL version includes the heading from the ABBREV version and then dumps each captured block in Hex and Character formats. For an example, see Figure 158 on page 344.
Figure 158. CICS trace output in the FULL format.

The first block is used for the feature trace information. It contains the name of the off-line formatting module and the short name used in the formatted heading line. The other 6 blocks are available for user data.

The SHORT version is a cross between the ABBREV and FULL versions.
Ensuring transaction rollback

If your application does not run to normal completion and there is no CICS transaction abend, take steps to ensure that transaction rollback (the backing out of any updates made by the malfunctioning application) takes place.

There are two ways to ensure that a transaction rollback occurs when an unhandled condition of severity 2 or greater is detected:

- Use the ABTERMENC runtime option with the ABEND suboption (ABTERMENC(ABEND))
- Use an assembler user exit that requests an abend for unhandled conditions of severity 2 or greater

The IBM-supplied assembler user exit for CICS (CEECXITA), available in the Language Environment SCEESAMP sample library, ensures that a transaction abend and rollback occur for all unhandled conditions of severity 2 or greater. For more information about the assembler user exit, see "Invoking the assembler user exit" on page 26 and z/OS Language Environment Programming Guide.

Finding data when Language Environment returns a nonzero return code

Language Environment does not write any messages to the CESE transient data queue. Table 48 shows the output generated when Language Environment returns a nonzero reason code to CICS and the location where the output appears.

<table>
<thead>
<tr>
<th>Output Message</th>
<th>Location</th>
<th>Issued By</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFHAC2206 14:43:54 LE03CC01 Transaction UTV2 has failed with abend AEC7. Resource backout was successful.</td>
<td>User's terminal</td>
<td>CICS</td>
</tr>
<tr>
<td>DFHAP12001I LE03CC01 A CICS request to the Language Environment has failed. Reason code '0012030'.</td>
<td>System console</td>
<td>CICS</td>
</tr>
<tr>
<td>DFHAC2236 06/05/91 14:43:48 LE03CC01 Transaction UTV2 abend AEC7 in routine UT2CVERI term P021 backout successful.</td>
<td>Transient data queue CSMT</td>
<td>CICS</td>
</tr>
</tbody>
</table>

Finding data when Language Environment abends internally

Language Environment does not write any messages to the CESE transient data queue. Table 49 shows the output generated when Language Environment abends internally and the location where the output appears:

<table>
<thead>
<tr>
<th>Output Message</th>
<th>Location</th>
<th>Issued By</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFHAC2206 14:35:24 LE03CC01 Transaction UTV8 has failed with abend 4095. Resource backout was successful.</td>
<td>User's terminal</td>
<td>CICS</td>
</tr>
<tr>
<td>CEE1000S LE INTERNAL abend. ABCODE = 00000FFF REASON = 00001234</td>
<td>System console</td>
<td>Language Environment</td>
</tr>
</tbody>
</table>
Table 49. Finding data when Language Environment abends internally (continued)

<table>
<thead>
<tr>
<th>Output Message</th>
<th>Location</th>
<th>Issued By</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFHAC2236 06/05/91 14:35:24 LE03CC01 Transaction UTV8 abend 4095 in routine</td>
<td>Transient data</td>
<td>CICS</td>
</tr>
<tr>
<td>UT8CVERI term P021 backout successful.</td>
<td>queue CSMT</td>
<td></td>
</tr>
</tbody>
</table>

Finding data when Language Environment abends from an EXEC CICS command

This section shows the output generated when an application abends from an EXEC CICS command and the location where the output appears. This error assumes the use of Language Environment runtime option TERMTHDACT(MSG).

Table 50. Finding data when Language Environment abends from an EXEC CICS command

<table>
<thead>
<tr>
<th>Output Message</th>
<th>Location</th>
<th>Issued By</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFHAC2206 14:35:34 LE03CC01 Transaction UTV8 has failed with abend AEI.</td>
<td>User's terminal</td>
<td>CICS</td>
</tr>
<tr>
<td>Resource backout was successful.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No message.</td>
<td>System console</td>
<td>CICS</td>
</tr>
<tr>
<td>DFHAC2236 06/05/91 14:35:17 LE03CC01 Transaction UTV9 abend AEI0 in routine</td>
<td>Transient data</td>
<td>CICS</td>
</tr>
<tr>
<td>UT9CVERI term P021 backout successful.</td>
<td>queue CSMT</td>
<td></td>
</tr>
<tr>
<td>P021UTV9 091156 143516 CEE3250C The System or User Abend AEI0 was issued.</td>
<td>Transient data</td>
<td>Language</td>
</tr>
<tr>
<td>CESE Environment</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Displaying and modifying runtime options with the CLER transaction

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

- Display the current runtime options in effect for the region.
- Modify the following subset of the region runtime options:
  - ALL31(ON|OFF)
  - CBLPSHPOP(ON|OFF)
  - CHECK(ON|OFF)
  - HEAPZONES(0-1024,QUIET | MSG | TRACE | ABEND)
  - INFOMSGFILTER(ON|OFF)
  - RPTOPTS(ON|OFF)
  - RPTSTG(ON|OFF)
  - TERMTHDACT(QUIET | MSG | TRACE | DUMP | UAONLY | UATRACE | UADUMP | UAIMM)
  - TRAP(ON|OFF)
- Write the current region runtime options to the CESE queue for printing.

The CLER transaction is conversational; it presents the user with commands for the terminal display. The runtime options that can be modified with this transaction are only in effect for the duration of the running region.
The CLER transaction must be defined in the CICS CSD (CICS System Definition file). The following definitions are required, and are in the Language Environment CEECCSD job in the SCEESAMP data set. Use the CEECCSD job to activate these definitions, or you must define them dynamically with the CICS CEDA transaction.

```
DEFINE PROGRAM(CEL4RTO) GROUP(CEE) LANGUAGE(ASSEMBLER) EXECKEY(CICS)
DEFINE MAPSET(CELEM) GROUP(CEE)
DEFINE MAPSET(CELHRH) GROUP(CEE)
DEFINE TRANS(CLER) PROG(CEL4RTO) GROUP(CEE)
```

**Note:** If the runtime option ALL31 is modified to OFF, the stack is forced to BELOW. When the stack is modified to BELOW, it will remain below for the duration of the region, even if you set ALL31 back to ON. A warning message, asking if you want to continue, is presented on the panel if the runtime option ALL31 is set to OFF or CBLPSHPOP, RPTOPTS, and RPTSTG are set to ON.

To send the runtime option report to the CESE queue for output display or printing, press PF10 on the panel which displays the runtime option report.

For detailed information on the use of CLER, select PF1 from the main menu that is displayed when the CLER transaction is invoked.
Part 3. Debugging Language Environment AMODE 64 applications

This part provides specific information for debugging applications written to make use of the memory address space above the 2 GB bar.
Chapter 10. Preparing your AMODE 64 application for debugging

This chapter describes options and features that you can use to prepare your AMODE 64 application for debugging. The following topics are covered:

- Compiler options for C, C++, PL/I
- Language Environment runtime options
- Use of storage in routines
- Options for modifying exception handling
- Assembler user exits
- Enclave termination behavior
- Language Environment feedback codes and condition tokens

Setting compiler options

The following sections discuss language-specific compiler options important to debugging routines in Language Environment. These sections cover only the compiler options that are important to debugging. For a complete list of compiler options, see the appropriate HLL publications.

The use of some compiler options (such as DEBUG) can affect the performance of your routine. You must set these options before you compile. In some cases, you might need to remove the option and recompile your routine before delivering your application.

**XL C and XL C++ compiler options for AMODE 64 applications**

When compiling an application using the LP64 compiler option, you cannot use the TEST compiler option. You must instead use the DEBUG(FORMAT(DWARF)) compiler option.

When the GONUMBER compiler option is used with LP64, it will produce executables with additional debug information. This is used by Language Environment to produce statement numbers in the Language Environment dump (CEEDUMP). Statement numbers in the CEEDUMP are also produced if the DEBUG compiler option or the c89 -g option is used.

For a detailed explanation of the debugging options for XL C/C++ and Inter-procedural Analysis (IPA), see "$OS XL C/C++ User’s Guide" and "$OS XL C/C++ Programming Guide".

Using Language Environment runtime options

Several runtime options affect debugging in Language Environment. The TEST runtime option, for example, can be used with a debugging tool to specify the level of control in effect for the debugging tool when the routine being initialized is started. The DYNDUMP, HEAPCHK, TERMTHDACT, TRACE, and TRAP options affect exception handling. The following Language Environment runtime options affect debugging. For a more detailed discussion of these runtime options, see "$OS Language Environment Programming Reference".
<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description of runtime option</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEDUMP</td>
<td>Specifies options to control the processing of the Language Environment dump report.</td>
</tr>
<tr>
<td>DYNDUMP</td>
<td>Provides a way to obtain IPCS readable dumps of user applications that would ordinarily be lost due to the absence of a SYSMDUMP, SYSUDUMP, or SYSABEND DD statement.</td>
</tr>
<tr>
<td>HEAPCHK</td>
<td>Determines whether additional heap check tests are performed.</td>
</tr>
<tr>
<td>HEAPZONES</td>
<td>Activates user heap overlay toleration and checking.</td>
</tr>
<tr>
<td>INFOMSGFILTER</td>
<td>Filters user specified informational messages from stderr. &lt;br&gt;Note: Affects only those messages generated by Language Environment and any routine that calls __le_msg_get_and_write(). Other routines that write to stderr, such as __le_msg_write(), do not have a filtering option.</td>
</tr>
<tr>
<td>PROFILE</td>
<td>Controls the use of an optional profiler tool, which collects performance data for the running application. When this option is in effect, the profiler is loaded and the debugger cannot be loaded. If the TEST option is in effect when PROFILE is specified, the profiler tool will not be loaded.</td>
</tr>
<tr>
<td>RPTOPTS</td>
<td>Causes a report to be produced which contains the runtime options in effect. See &quot;Determining runtime options in effect&quot; below.</td>
</tr>
<tr>
<td>RPTSTG</td>
<td>Generates a report of the storage used by an enclave. See &quot;Controlling storage allocation&quot; on page 354.</td>
</tr>
<tr>
<td>STORAGE</td>
<td>Specifies that Language Environment initializes all heap and stack storage to a user-specified value.</td>
</tr>
<tr>
<td>TERMTHDACT</td>
<td>Controls response when an enclave terminates due to an unhandled condition of severity 2 or greater.</td>
</tr>
<tr>
<td>TEST</td>
<td>Specifies the conditions under which a debugging tool assumes control.</td>
</tr>
<tr>
<td>TRACE</td>
<td>Activates Language Environment runtime library tracing and controls the size of the trace table, the type of trace, and whether the trace table should be dumped unconditionally upon termination of the application.</td>
</tr>
<tr>
<td>TRAP</td>
<td>When TRAP is set to ON, Language Environment traps routine interrupts and abends, and optionally prints trace information or invokes a user-written exception handling routine. With TRAP set to OFF, the operating system handles all interrupts and abends. You should generally set TRAP to ON, or your runtime results can be unpredictable.</td>
</tr>
</tbody>
</table>

### Determining runtime options in effect

The runtime options in effect at the time the routine is run can affect routine behavior. Use RPTOPTS(ON) to generate an options report in the Language Environment message file when your routine terminates. The options report lists runtime options, and indicates where they were set. Figure 159 on page 353 shows a sample options report.
Understanding the HEAPZONES and HEAPCHK runtime options

The HEAPZONES and HEAPCHK runtime options are useful for debugging overlay damage problems that occur in the user heap. Though similar in that both options can be used for debugging purposes, the runtime options activate very different behavior in the runtime when specified.

HEAPZONES is a lightweight mechanism that detects heap overlay damage only during the freeing of an element. It looks for damage in the heap check zone of the freed element only.

Selecting a non-quiet output option causes HEAPZONES to display information about the damaged heap element. When messaging is requested, the address of the damaged element along with information specific to the heap check zone are included in the message. Depending on the type of damage, the value of the heap check zone is displayed. The data area of the damaged location is displayed following any issued informational messages. This runtime option can also be used as a mechanism to tolerate heap overlay damage by simply requesting no output (QUIET).
Depending on the size of the heap check zone and the number of allocation requests, the user may notice a significant amount of extra storage being used by the application. Performance may be affected due to the overhead of examining each heap check zone.

HEAPCHK investigates the entire user heap for damage during heap related calls at a frequency based on the specified settings in the option. Because HEAPCHK will traverse the entire user heap, a slow down in application performance will occur. Information about HEAPCHK diagnostic output is discussed in Chapter 3, “Using Language Environment debugging facilities,” on page 39.

When deciding which runtime option is better suited to use with your application, consider the differences between HEAPZONES and HEAPCHK relating to performance, storage usage, and time of damage detection. Although both runtime options affect performance, an application that chooses HEAPCHK will perform slower than an application that chooses HEAPZONES. If storage usage is a concern, HEAPCHK will not consume extra amounts of storage in the manner that HEAPZONES will. Determining when heap damage has occurred may be simpler to accomplish if HEAPCHK is chosen because of the frequency and scope of its analysis.

For more information about the HEAPZONES and HEAPCHK runtime options, see z/OS Language Environment Programming Reference.

### Controlling storage allocation

The following runtime options control storage allocation:

- HEAP64
- HEAPPOOLS
- HEAPPOOLS64
- IOHEAP64
- LIBHEAP64
- STACK64
- THREADSTACK64

z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode provides useful tips to assist with the tuning process. Appropriate tuning is necessary to avoid performance problems.

To generate a report of the storage a routine (or more specifically, an enclave) used during its run, specify the RPTSTG(ON) runtime option. The storage report, generated during enclave termination provides statistics that can help you understand how space is being consumed as the enclave runs. If storage management tuning is desired, the statistics can help you set the corresponding storage-related runtime options for future runs. Figure 160 on page 355 shows a sample storage report.
STACK64 statistics:
Initial size: 1M
Increment size: 1M
Maximum used by all concurrent threads: 1M
Largest used by any thread: 1M
Number of increments allocated: 0

THREADSTACK64 statistics:
Initial size: 1M
Increment size: 1M
Maximum used by all concurrent threads: 0M
Largest used by any thread: 0M
Number of increments allocated: 0

64bit User HEAP statistics:
Initial size: 1M
Increment size: 1M
Total heap storage used: 983808
Suggested initial size: 1M
Successful Get Heap requests: 11
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0

31bit User HEAP statistics:
Initial size: 32768
Increment size: 32768
Total heap storage used (sugg. initial size): 243352
Successful Get Heap requests: 58
Successful Free Heap requests: 0
Number of segments allocated: 9
Number of segments freed: 0

24bit User HEAP statistics:
Initial size: 4096
Increment size: 4096
Total heap storage used (sugg. initial size): 0
Successful Get Heap requests: 0
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0

64bit Library HEAP statistics:
Initial size: 1M
Increment size: 1M
Total heap storage used: 3795584
Suggested initial size: 4M
Successful Get Heap requests: 384
Successful Free Heap requests: 337
Number of segments allocated: 2
Number of segments freed: 0

31bit Library HEAP statistics:
Initial size: 16384
Increment size: 8192
Total heap storage used (sugg. initial size): 0
Successful Get Heap requests: 0
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0

Figure 160. 64–bit storage report (Part 1 of 4)
### 24bit Library HEAP statistics:
- Initial size: 8192
- Increment size: 4096
- Total heap storage used (sugg. initial size): 0
- Successful Get Heap requests: 0
- Successful Free Heap requests: 0
- Number of segments allocated: 0
- Number of segments freed: 0

### 64bit I/O HEAP statistics:
- Initial size: 1M
- Increment size: 1M
- Total heap storage used: 0
- Suggested initial size: 1M
- Successful Get Heap requests: 0
- Successful Free Heap requests: 0
- Number of segments allocated: 0
- Number of segments freed: 0

### 31bit I/O HEAP statistics:
- Initial size: 12288
- Increment size: 8192
- Total heap storage used (sugg. initial size): 9616
- Successful Get Heap requests: 27
- Successful Free Heap requests: 19
- Number of segments allocated: 1
- Number of segments freed: 0

### 24bit I/O HEAP statistics:
- Initial size: 4096
- Increment size: 4096
- Total heap storage used (sugg. initial size): 3032
- Successful Get Heap requests: 14
- Successful Free Heap requests: 6
- Number of segments allocated: 1
- Number of segments freed: 0

### HEAPPools Statistics:
- Pool 1 size: 8 Get Requests: 0
- Pool 2 size: 32 Get Requests: 1
- Successful Get Heap requests: 17-24
- Pool 3 size: 128 Get Requests: 0
- Pool 4 size: 256 Get Requests: 0
- Pool 5.1 size: 1024 Get Requests: 225
- Pool 5.2 size: 1024 Get Requests: 0
- Pool 5.3 size: 1024 Get Requests: 0
- Successful Get Heap requests: 273-280
- Pool 6 size: 2048 Get Requests: 225
- Requests greater than the largest cell size: 0

### HEAPPools Summary:

<table>
<thead>
<tr>
<th>Specified Cell Size</th>
<th>Element Extent</th>
<th>Cells Per Extent</th>
<th>Allocated Maximum Cells Used</th>
<th>Cells In Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td>10</td>
<td>409</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>40</td>
<td>10</td>
<td>163</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>136</td>
<td>10</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>264</td>
<td>10</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1032</td>
<td>10</td>
<td>4</td>
<td>57</td>
</tr>
<tr>
<td>1024</td>
<td>1032</td>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1032</td>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>2056</td>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Suggested Percentages for current Cell Sizes:
- HEAPP(ON,8,1,32,1,128,1,256,1,(1024,3),90,2048,1,0)
- Suggested Cell Sizes:
  - HEAPP(ON,24,280,,2048,0)

---

*Figure 161. 64–bit storage report (Part 2 of 4)*
**HEAPPOOLS64 Statistics:**

<table>
<thead>
<tr>
<th>Pool</th>
<th>Size</th>
<th>Get Requests</th>
<th>Successful Get Heap Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1-8</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>240</td>
<td>9-16, 17-24, 25-32</td>
</tr>
<tr>
<td>3</td>
<td>128</td>
<td>125</td>
<td>33-40, 41-48, 57-64, 65-72, 73-80, 81-88, 89-96, 105-112, 113-120, 121-128</td>
</tr>
<tr>
<td>5.1</td>
<td>1024</td>
<td>2</td>
<td>129-136</td>
</tr>
<tr>
<td>5.2</td>
<td>1024</td>
<td>2</td>
<td>137-144</td>
</tr>
<tr>
<td>5.3</td>
<td>1024</td>
<td>0</td>
<td>145-152, 153-160</td>
</tr>
<tr>
<td>6</td>
<td>2048</td>
<td>2</td>
<td>161-168, 169-176</td>
</tr>
<tr>
<td>7</td>
<td>3072</td>
<td>2</td>
<td>177-184, 185-192</td>
</tr>
<tr>
<td>8</td>
<td>4096</td>
<td>1</td>
<td>193-200, 201-208</td>
</tr>
<tr>
<td>9</td>
<td>8192</td>
<td>0</td>
<td>209-216, 217-224</td>
</tr>
<tr>
<td>10</td>
<td>16384</td>
<td>0</td>
<td>225-232, 233-240</td>
</tr>
<tr>
<td>11</td>
<td>32768</td>
<td>0</td>
<td>249-256, 251-256</td>
</tr>
<tr>
<td>12</td>
<td>65536</td>
<td>0</td>
<td>251-256</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Requests greater than the largest cell size: 0</td>
</tr>
</tbody>
</table>

![Figure 162. 64-bit storage report (Part 3 of 4)](image-url)
Storage statistics for AMODE 64 applications

The statistics for initial and incremental allocations of storage types that have a corresponding runtime option differ from the runtime option settings when their values have been rounded up by the implementation, or when allocations larger than the amounts specified were required during execution. See the descriptions of the runtime options in $\textit{z/OS Language Environment Programming Reference}$ for information about rounding.

Stack storage statistics for AMODE 64 applications

Language Environment stack storage is managed at the thread level—each thread has its own stack-type resources.

STACK64 and THREADSTACK64 statistics:

- Initial size—the actual size of the initial stack area assigned to each thread. If a pthread-attributes-table is provided on the invocation of pthread-create, the stack size specified in the pthread-attributes-table takes precedence over the stack runtime options.
- Increment size—the size of each incremental stack area made available, as determined by the increment portion of the corresponding runtime option.
- Maximum used by all concurrent threads—the maximum amount allocated in total at any one time by all concurrently executing threads.
- Largest used by any thread—the largest amount allocated ever by any single thread.
- Number of increments allocated—the number of incremental segments allocated by all threads.

Determining the applicable threads: If the application is not a multithreading application, the STACK64 statistics are for the one and only thread that executed, and the THREADSTACK64 statistics are all zero.

<table>
<thead>
<tr>
<th>Specified</th>
<th>Element</th>
<th>Cells Per Extents</th>
<th>Maximum Cells Used</th>
<th>Cells In Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Size</td>
<td>Size</td>
<td>Extents</td>
<td>Allocated</td>
<td>Use</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>------------------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>4000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>48</td>
<td>2000</td>
<td>1</td>
<td>226</td>
</tr>
<tr>
<td>128</td>
<td>144</td>
<td>700</td>
<td>1</td>
<td>83</td>
</tr>
<tr>
<td>256</td>
<td>272</td>
<td>350</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1024</td>
<td>1040</td>
<td>34</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1024</td>
<td>1040</td>
<td>34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1040</td>
<td>34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>2064</td>
<td>50</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3072</td>
<td>3088</td>
<td>50</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4096</td>
<td>4112</td>
<td>50</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8192</td>
<td>8208</td>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>16400</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>32784</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>65536</td>
<td>65552</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Suggested Cell Sizes:

HP64(ON, 40,80,96,128,168,224,288,528,720,1648,2112,3688)

Largest number of threads concurrently active: 6

End of Storage Report

Figure 163. 64-bit storage report (Part 4 of 4)
If the application is a multithreading application, and THREADSTACK64 was not suppressed, the STACK64 statistics are for the initial thread (IPT), and the THREADSTACK64 statistics are for the other threads. However, if THREADSTACK64 was suppressed, the STACK64 statistics are for all of the threads, initial and other.

**Allocating stack storage:** The allocation of the stack for each thread, including the initial processing thread (IPT), is part of a storage request to the system when the thread is first created. Other storage, not part of the stack, is also acquired at this time. These storage allocations are not shown in the storage report. The size of the stack portion of this storage is the stack maximum size plus a one megabyte (1M) guard area. After allocation, the guard area follows the stack initial size and runs through the end of the stack maximum size plus the 1M guard area. Increments to the stack for each thread do not result in additional storage requests to the system. They result in the movement of the beginning of the guard area no further than the maximum size of the stack. The stack initial, increment, and maximum sizes are controlled through the STACK64 and THREADSTACK64 runtime options.

**Heap storage statistics**
Language Environment heap storage is managed at the enclave level. Each enclave has its own heap type resources, which are shared by the threads that execute within the enclave. The heap resources have 64-bit, 31-bit, and 24-bit addressable areas, each of which can be tuned separately.

**HEAP64, LIBHEAP64, and IOHEAP64 statistics:**
- Initial size—the default initial allocation, as specified by the corresponding runtime option.
- Increment size—the minimum incremental allocation, as specified by the corresponding runtime option.
- Total heap storage used—the largest total amount used by the enclave at any one time.
- Successful Get Heap requests—the number of get heap requests.
- Successful Free Heap requests—the number of free heap requests.
- Number of segments allocated—the number of incremental segments allocated.
- Number of segments freed—the number of incremental segments individually freed.

The number of Free Heap requests could be less than the number of Get Heap requests if the pieces of heap storage acquired by individual Get Heap requests were not explicitly freed, but were freed implicitly during enclave termination. The number of incremental segments individually freed could be less than the number allocated if the segments were not explicitly freed, but were freed implicitly during enclave termination. The initial segment is included in **Number of segments allocated** for each 31-bit and 24-bit addressable heap resource, and for the 64-bit addressable IOHEAP64 resource. A disposition of KEEP always causes 0 to be reported for the **Number of segments freed**. These statistics, in all cases, specify totals for the entire enclave.

**Heap pools storage statistics**
The HEAPPOOLS and HEAPPOOLS64 runtime options for C/C++ applications only controls usage of the heap pools storage algorithm at the enclave level. The heap pools algorithm allows for the definition of one to twelve heap pools, each consisting of a number of storage cells of a specified length. For further details regarding heap pools storage statistics in the storage report, see "Language Environment storage report with heap pools statistics" on page 503.
Modifying exception handling behavior

Setting the exception handling behavior of your routine affects the response that occurs when the routine encounters an error. You can modify exception handling behavior in the following ways:

- Application program interfaces (API)
- User-written exception handlers
- POSIX functions (used to specifically set signal actions and signal masks)

Language Environment application program interfaces (API)

You can use the following APIs to modify exception handling:

<table>
<thead>
<tr>
<th>Function name</th>
<th>API description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cabend()</td>
<td>Terminates an enclave using an abend.</td>
</tr>
<tr>
<td>__le_cib_get()</td>
<td>Returns a pointer to a condition information block (CIB) associated with a given condition token. The CIB contains detailed information about the condition.</td>
</tr>
<tr>
<td>__set_exception_handler()</td>
<td>Activates a routine to handle an exception.</td>
</tr>
<tr>
<td>__reset_exception_handler()</td>
<td>Removes handling of an exception by any routine.</td>
</tr>
</tbody>
</table>

Language Environment runtime options

The following Language Environment runtime options can affect your routine's exception handling behavior:

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description of runtime option</th>
</tr>
</thead>
</table>
| TERMTHDACT       | Sets the level of information that is produced when a condition of severity 2 or greater remains unhandled within the enclave. The possible parameter settings for different levels of information are:  
|                  |   • QUIET for no information  
|                  |   • MSG for message only  
|                  |   • TRACE for message and a traceback  
|                  |   • DUMP for message, traceback, and Language Environment dump  
|                  |   • UAONLY for message and a system dump of the user address space  
|                  |   • UATRACE for message, Language Environment dump with traceback information only, and a system dump of the user address space  
|                  |   • UADUMP for message, traceback, Language Environment dump, and system dump  
|                  |   • UAIMM for a system dump of the user address space of the original abend or program interrupt prior to the Language Environment condition manager processing the condition. |
| TRAP(ON)         | Fully enables the Language Environment exception handler. This causes the Language Environment exception handler to intercept error conditions and routine interrupts.  
|                  | When TRAP(ON, NOSPIE) is specified, Language Environment handles all program interrupts and abends through an ESTAE. Use this feature when you do not want Language Environment to issue an ESPIE macro.  
<p>|                  | During normal operation, you should use TRAP(ON) when running your applications.  |</p>
<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description of runtime option</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAP(OFF)</td>
<td>Disables the Language Environment condition handler from handling abends and program checks/interrupts. ESPIE is not issued with TRAP(OFF).</td>
</tr>
</tbody>
</table>

Specify TRAP(OFF) when you do not want Language Environment to issue an ESPIE.

When TRAP(OFF), TRAP(OFF,SPIE), or TRAP(OFF,NOSPIE) is specified and either a program interrupt or abend occurs, the user exit for termination is ignored.

TRAP(OFF) can cause several unexpected side effects. It is not supported in AMODE 64 production execution.

For further information, see the TRAP runtime option in z/OS Language Environment Programming Reference.

Customizing exception handlers

User-written exception handlers permit you to customize exception handling for certain conditions. You can register a user-written exception handler for the current stack frame by using the __set_exception_handler() API. For more information about user-written exception handlers and the Language Environment condition manager, see z/OS XL C/C++ Programming Guide.

Using condition information

If a condition that might require attention occurs while an application is running, Language Environment builds a condition token. The condition token contains 16 bytes (128 bits) of information about the condition that Language Environment or your routines can use to respond appropriately. Each condition is associated with a single Language Environment runtime message. You can use this condition information in two ways:

- To specify the feedback code parameter when calling Language Environment services (see “Using the feedback code parameter”).
- To code a symbolic feedback code in a user-written exception handler (see “Using the symbolic feedback code” on page 363).

Using the feedback code parameter

The feedback code is an optional parameter of the Language Environment APIs. For C/C++ applications, this parameter is optional. For more information about feedback codes and condition tokens, see z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode.

When you provide the feedback code (fc) parameter, the API in which the condition occurs sets the feedback code to a specific value called a condition token.

The condition token does not apply to asynchronous signals. For a discussion of the distinctions between synchronous signals and asynchronous signals with POSIX(ON), see z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode.

When you do not provide the fc parameter, any nonzero condition is signaled and processed by Language Environment exception handling routines. If you have registered a user-written exception handler, Language Environment passes control...
to the handler, which determines the next action to take. If the condition remains unhandled, Language Environment writes a message to stderr. The message is the translation of the condition token into English (or another supported national language).

Language Environment provides APIs that can be used to convert condition tokens to routine variables, messages, or signaled conditions. The following table lists these Language Environment APIs and their functions. For more information on these APIs, see z/OS XL C/C++ Programming Guide.

<table>
<thead>
<tr>
<th>API name</th>
<th>API description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__le_msg_write()</td>
<td>Writes a message string to stderr</td>
</tr>
<tr>
<td>__le_msg_get_and_write()</td>
<td>Takes a message associated with a condition and writes it to stderr</td>
</tr>
<tr>
<td>__le_msg_get()</td>
<td>Retrieves, formats, and stores message data for a condition</td>
</tr>
<tr>
<td>__le_msg_add_insert()</td>
<td>Creates a message insert</td>
</tr>
</tbody>
</table>

There are two types of condition tokens. Case 1 condition tokens contain condition information, including the Language Environment message number. All Language Environment APIs and most application routines use case 1 condition tokens. Case 2 condition tokens contain condition information and a user-specified class and cause code. Application routines, user-written exception handlers, assembler user exits, and some operating systems can use case 2 condition tokens.

For Case 1 condition tokens, Condition_ID is:

```
Condition_ID 32 - 33
Case Number 34 - 36
Severity Number 37 - 39
Control Code 40 - 63
Facility_ID 64 - 127
```

For Case 2 condition tokens, Condition_ID is:

```
Condition_ID 0 - 15
Case Number 16 - 31
Severity Number 32 - 33
Message Number 34 - 36
Cause Code 40 - 63
Facility_ID 64 - 127
```

A symbolic feedback code represents the first 8 bytes of a condition token. It contains the Condition_ID, Case Number, Severity Number, Control Code, and Facility_ID, whose bit offsets are indicated.

**Figure 164. Language Environment condition token**

For example, in the condition token: X'0003032D 59C3C5C5 00000000 00000000'
- X'0003' is severity.
- X'032D' is message number 813.
- X'59' are hexadecimal flags for case, severity, and control.
- X'C3C5C5' is the CEE facility ID.
- X'00000000 00000000' is the instance specific information (ISI). (In this case, no ISI was provided.)

If a Language Environment traceback or dump is generated while a condition token is being processed or when a condition exists, Language Environment writes the runtime message to the condition section of the traceback or dump. If a condition is detected when a Language Environment API is invoked without a
feedback code, the condition token is passed to the Language Environment condition manager. If a condition is severity 0 or 1, Language Environment resumes without issuing a message. For conditions of severity 2 or greater, Language Environment issues a message and terminates. For a list of Language Environment runtime messages and corrective information, see z/OS Language Environment Runtime Messages.

If a second condition is raised while Language Environment is attempting to handle a condition, the message CEE0374C CONDITION = <message no.> is displayed using a write-to-operator (WTO). The message number in the CEE0374C message indicates the original condition that was being handled when the second condition was raised. This can happen when a critical error is signaled (for example, when internal control blocks are damaged).

If the output for this error message appears several times in sequence, the conditions appear in order of occurrence. Correcting the earliest condition can cause your application to run successfully.

**Using the symbolic feedback code**

The symbolic feedback code represents the first 8 bytes of a 16-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 exception handlers to screen for a given condition, even if it occurs at different locations in an application. For more details on symbolic feedback codes, see z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode.
Chapter 11. Classifying AMODE 64 application errors

This chapter describes errors that commonly occur in Language Environment AMODE 64 applications. It also explains how to use runtime messages and abend codes to obtain information about errors in your application.

Identifying problems in routines

The following sections describe how you can identify errors in Language Environment routines. Included are common error symptoms and solutions.

Language Environment module names

You can identify Language Environment-supplied module elements by any of the following three-character prefixes:

- CEE (Language Environment)
- CEL (Language Environment)
- EDC (C/C++)

Module elements or text files with other prefixes are not part of the Language Environment product for AMODE 64 applications.

Common errors in routines

These common errors have simple solutions:

- If you receive abend U4093, reason X'224' (548 decimal), then make sure you use MEMLIMIT to allow access to above the 2 GB bar. For more information, see z/OS MVS Programming: Extended Addressability Guide.
- If you do not have enough virtual storage, increase your region size or decrease your storage usage (stack size) by using the storage-related runtime options and callable services. (See "Controlling storage allocation" on page 354 for information about using storage in routines.)
- If you do not have enough disk space, increase your disk allocation.
- If executable files are not available, check your executable library to ensure that they are defined. For example, check your STEPLIB or JOBLIB definitions.

If your error is not caused by any of the items listed above, examine your routine or routines for changes since the last successful run. If there have been changes, review these changes for errors that might be causing the problem. One way to isolate the problem is to branch around or comment out recent changes and rerun the routine. If the run is successful, the error can be narrowed to the scope of the changes.

Changes in optimization levels, addressing modes, and input/output file formats can also cause unanticipated problems in your routine.

In most cases, generated condition tokens or runtime messages point to the nature of the error. The runtime messages offer the most efficient corrective action. To help you analyze errors and determine the most useful method to fix the problem, Table 1 on page 366 lists common error symptoms, possible causes, and programmer responses.
## Table 51. Common error symptoms, possible causes, and programmer responses

<table>
<thead>
<tr>
<th>Error Symptom</th>
<th>Possible Cause</th>
<th>Programmer Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbered runtime message appears</td>
<td>Condition raised in routine</td>
<td>For any messages you receive, read the [Interpreting runtime messages] below.</td>
</tr>
</tbody>
</table>
| User abend code < 4000 | - A non-Language Environment abend occurred  
- The assembler user exit requested an abend for an unhandled condition of severity \( \geq 2 \) | See the Language Environment abend codes in \[z/OS Language Environment Runtime Messages\] below.  
Check for a subsystem-generated abend or a user-specified abend. |
| User abend code \( \geq 4000 \) | - Language Environment detected an error and could not proceed  
- An unhandled software-raised condition occurred  
- The assembler user exit requested an abend for an unhandled condition of severity 4 | For any abends you receive, read the appropriate explanation listed in the abend codes section of \[z/OS Language Environment Runtime Messages\]. |
| System abend with TRAP(OFF) | Cause depends on type of malfunction | Respond appropriately. See the messages and codes book of the operating system. |
| System abend with TRAP(ON) | System-detected error | See the messages and codes book of the operating system. |
| No response (wait/loop) | Application logic failure | Check routine logic. |
| Unexpected message (message received was not from most recent service) | Condition caused by something related to current service | Generate a traceback using cdump() or ctrace(). |
| Incorrect output | Incorrect file definitions, storage overlay, incorrect routine mask setting, references to uninitialized variables, data input errors, or application routine logic error | Correct the appropriate parameters. |
| No output | Incorrect ddname or file definitions | Correct the appropriate parameters. |
| Nonzero return code from enclave | The return code was issued by the application routine | Check the application for the meaning of the return code. |

### Interpreting runtime messages

The first step in debugging your routine is to look up any runtime messages. Runtime messages are written to the C stderr stream. Runtime messages provide users with additional information about a condition, and possible solutions for any errors that occurred. They can be issued by Language Environment common routines or language-specific runtime routines and contain a message prefix, message number, severity code, and descriptive text.

In the following example Language Environment message:

CEE3206S The system detected a specification exception (System Completion Code=0C6).

- The message prefix is CEE.
- The message number is 3206.
- The severity code is S.
- The message text is “The system detected a specification exception (System Completion Code=0C6)”.

366  
\[z/OS Language Environment Debugging Guide\]
Language Environment messages can appear even though you made no explicit calls to Language Environment services. C/C++ runtime library routines commonly use the Language Environment services. This is why you can see Language Environment messages even when the application routine does not directly call common runtime services.

**Message prefix**
The message prefix indicates the Language Environment component that generated the message. The message prefix is the first three characters of the message number and is also the facility ID in the condition token. The messages for the various components can be found in [z/OS Language Environment Runtime Messages](#).

<table>
<thead>
<tr>
<th>Message Prefix</th>
<th>Language Environment Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE</td>
<td>Common run time</td>
</tr>
<tr>
<td>EDC</td>
<td>C/C++ run time</td>
</tr>
</tbody>
</table>

**Message number**
The message number is the 4-digit number following the message prefix. Leading zeros are inserted, if the message number is less than four digits. It identifies the condition raised and references additional condition and programmer response information.

**Severity code**
The severity code is the letter following the message number and indicates the level of attention called for by the condition. Messages with severity “I” are informational messages and do not usually require any corrective action. In general, if more than one runtime message appears, the first noninformational message indicates the problem. For a complete list of severity codes, severity values, condition information, and default actions, see [z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode](#).

**Message text**
The message text provides a brief explanation of the condition.

### Understanding abend codes
Under Language Environment, abnormal terminations generate abend codes. There are two types of abend codes: 1) user abends (Language Environment and user-specified) and 2) system abends. User abends follow the format of Udddd, where dddd is a decimal user abend code. System abends follow the format of Shhh, where lhh is a hexadecimal abend code. Language Environment abend codes are usually in the range of 4000 to 4095. However, some subsystem abend codes can also fall in this range. User-specified abends use the range of 0 to 3999.

Example abend codes are:
User (Language Environment) abend code:U4041
User-specified abend code:U0005
System abend code:S80A

The Language Environment API __cabend() terminates your application with an abend. You can set the clean_up parameter value to determine how the abend is
processed and how Language Environment handles the raised condition. For more information about __cabend() and clean_up, see z/OS XL C/C++ Runtime Library Reference.

User abends

If you receive a Language Environment abend code, see z/OS Language Environment Runtime Messages for a list of abend codes, error descriptions, and programmer responses.

System abends

If you receive a system abend code, look up the code and the corresponding information in the publications for the system you are using. When a system abend occurs, the operating system can generate a system dump. System dumps are written to ddname SYSMDUMP, SYSABEND, or SYSUDUMP. If the DYNDUMP runtime option is used in combination with the TERMTHDACT runtime option, the system dump can be written without the ddname specified. System dumps show the memory state at the time of the condition. See “Generating a system dump” on page 386 for more information about system dumps.
Chapter 12. Using Language Environment AMODE 64 debugging facilities

This section describes methods of debugging AMODE 64 routines in Language Environment. Currently, most problems in Language Environment and member language routines can be determined through the use of a debugging tool or through information provided in the Language Environment dump.

Debugging tools

You can use dbx to debug Language Environment applications. z/OS UNIX System Services Command Reference has information on dbx subcommands, while z/OS UNIX System Services Programming Tools contains usage information.

Language Environment dumps

The following sections provide information about using the Language Environment dump service, and describe the contents of the Language Environment dump.

Generating a Language Environment dump with TERMTHDACT

The TERMTHDACT runtime option produces a dump during program checks or abnormal terminations. You must use TERMTHDACT(DUMP) in conjunction with TRAP(ON) to generate a Language Environment dump. You can use TERMTHDACT to produce a traceback, Language Environment dump, or user address space dump when a thread ends abnormally because of an unhandled condition of severity 2 or greater. If this is the last thread in the process, the enclave goes away. A thread terminating in a non-POSIX environment is analogous to an enclave terminating. For information on enclave termination, see z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode.

The TERMTHDACT suboptions QUIET, MSG, TRACE, DUMP, UANOY, UATRACE, UADUMP, and UAIMM control the level of information available. Following are the suboptions, the levels of information produced, and the destination of each.

<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIET</td>
<td>No information</td>
<td>No destination.</td>
</tr>
<tr>
<td>MSG</td>
<td>Message</td>
<td>Stderr</td>
</tr>
<tr>
<td>TRACE</td>
<td>Message and Language Environment dump containing only a traceback</td>
<td>Message goes to stderr. Traceback goes to CEEDUMP file.</td>
</tr>
<tr>
<td>DUMP</td>
<td>Message and complete Language Environment dump</td>
<td>Message goes to stderr. Language Environment dump goes to CEEDUMP file.</td>
</tr>
</tbody>
</table>

Table 52. TERMTHDACT suboptions, level of information, and destinations
<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAONLY</td>
<td>SYSMDUMP, SYSABEND dump, or SYSUDUMP depending on the DD card used in the JCL in z/OS. You will get a system dump of your user address space if the appropriate DD statement is used. <strong>Note:</strong> A Language Environment dump is not generated.</td>
<td>Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified.</td>
</tr>
<tr>
<td>UATRACE</td>
<td>Message, Language Environment dump containing only a traceback, and a system dump of the user address space</td>
<td>Message goes to stderr. Traceback goes to CEEDUMP file. Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified.</td>
</tr>
<tr>
<td>UADUMP</td>
<td>Message, Language Environment dump, and SYSMDUMP, SYSABEND dump, or SYSUDUMP depending on the DD card used in the JCL in z/OS.</td>
<td>Message goes to stderr. Language Environment dump goes to CEEDUMP file. Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified.</td>
</tr>
<tr>
<td>UAIMM</td>
<td>Language Environment generates a system dump of the original abend/program interrupt of the user address space. You will get a system dump of your user address space if the appropriate DD statement is used. After the dump is taken by the operating system, Language Environment condition manager continues processing.</td>
<td>Message goes to stderr. User address space dump goes to ddname specified for z/OS.</td>
</tr>
</tbody>
</table>

The TRACE and UATRACE suboptions of TERMTHDACT use these dump options:
- CONDITION
- ENCLAVE(ALL)
- FILES
- FNAME(CEEDUMP)
- GENOPTS
- NOBLOCKS
- NOENTRY
- NOSTORAGE
- STACKFRAME(ALL)
- THREAD(ALL)
- TRACEBACK
- VARIABLES

The DUMP and UADUMP suboptions of TERMTHDACT use these dump options:
- BLOCKS
- CONDITION
- ENCLAVE(ALL)
- FILES
- FNAME(CEEDUMP)
- GENOPTS
- NOENTRY
- STACKFRAME(ALL)
- STORAGE
- THREAD(ALL)
Considerations for setting TERMTHDACT options

Review the following considerations before setting TERMTHDACT runtime options. For more information, see z/OS Language Environment Programming Reference.

- z/OS UNIX Considerations
  - The TERMTHDACT option applies when a thread terminates abnormally. Abnormal termination of a single thread causes termination of the entire enclave. If an unhandled condition of severity 2 or higher percolates beyond the first routine’s stack frame, the enclave terminates abnormally.
  - If an enclave terminates due to a POSIX default signal action, then TERMTHDACT applies to conditions that result from software signals, program checks, or abends.
  - If running under a shell and Language Environment generates a system dump, then a core dump is generated to a file based on the kernel environment variable, _BPXK_MDUMP.

- Preinitialized Environments for Authorized Programs Considerations
  - The TERMTHDACT suboptions TRACE, DUMP, UADUMP, UATRACE are overridden to UAONLY.
  - For UAONLY, a U4039 abend is generated and an SVC dump of the U4039 abend with the following title is taken:

```
COMPON=CEL,COMPID=568819801,ISSUER=CELAFRR ,MODULE=CELAEICT+????, ABEND=U4039,REASON=00000000
```

  - For UAONLY, an SVC dump of the original abend/program interrupt with the following title is taken (the ABEND and REASON values are those of the original abend/program interrupt):

```
COMPON=CEL,COMPID=568819801,ISSUER=CELAFRR ,MODULE=CELAEICT+????, ABEND=S00C9,REASON=00000009
```

Generating a Language Environment dump with language-specific functions

C/C++ routines can use the functions cdump(), csnap(), and ctrace() to produce a Language Environment dump. For more information on these functions, see “Generating a Language Environment dump of a C/C++ routine” on page 478.

Understanding the Language Environment dump

The Language Environment dump service generates output of data and storage from the Language Environment runtime environment on an enclave basis. This output contains the information needed to debug most basic routine errors.

Figure 168 on page 374 illustrates a dump for enclave main. The example shows full use of the TERMTHDACT dump options. Ellipses are used to summarize some sections of the dump and information regarding unhandled conditions may not be present at all. Sections of the dump are numbered to correspond with the descriptions given in Figure 165 on page 372.
The CEE3DMP was generated by the C program CELQSAMP shown in Figure 165. CELQSAMP uses the DLL CELQDLL shown in Figure 167 on page 373.

```c
#include <stdio.h>
#include <stdlib.h>
#include <dll.h>

typedef void* FUNC(void *
);

pthread_mutex_t mut;
pthread_t thread[2];
int threads_joined = 0;
char * t1 = "Thread 1";
char * t2 = "Thread 2";

/* thread_func: Invoked via pthread_create. */
void *thread_func(void *parm)
{
    printf(">>> Thread_func: %s locking mutex\n",parm);
    pthread_mutex_lock(&mut);
    pthread_mutex_unlock(&mut);
    printf(">>> Thread_func: %s exiting\n",parm);
    pthread_exit(NULL);
}

main()
{
    dllhandle * handle;
    FUNC * fp;
    FILE* fp1;
    FILE* fp2;

    printf("Load DLL...\n");
    handle = dllload("CELQDLL");
    if (handle == NULL) {
        perror("Could not load DLL CELQDLL");
        exit(106);
    }

    printf("Query DLL...\n");
    fp = (FUNC *)dllqueryfn(handle,"div_zero");
    if (fp == NULL) {
        perror("Could not find thread_func");
        exit(107);
    }

    printf("Init MUX...\n");
    if (pthread_mutex_init(&mut, NULL) == -1) {
        perror("Init of mut failed");
        exit(101);
    } printf("Lock Mux Lock...\n");
    if (pthread_mutex_lock(&mut) == -1) {
        perror("Lock of mut failed");
        exit(102);
    }
}
```

Figure 165. The C program CELQSAMP (AMODE 64) (Part 1 of 2)
printf("Create 1st thread...
");
if (pthread_create(&thread[0],NULL,thread_func,(void *)t1) == -1) {
    perror("Could not create thread #1");
    exit(103);
}

printf("Create 2nd thread...
");
if (pthread_create(&thread[1],NULL,thread_func,(void *)t2) == -1) {
    perror("Could not create thread #2");
    exit(104);
}

printf("Write to some files...
");
fp1 = fopen("myfile.data", "w");
if (!fp1) {
    perror("Could not open myfile.data for write");
    exit(109);
}

fprintf(fp1, "record 1\n");
fprintf(fp1, "record 2\n");
fprintf(fp1, "record 3\n");

fp2 = fopen("memory.data", "wb,type=memory");
if (!fp2) {
    perror("Could not open memory.data for write");
    exit(112);
}

fprintf(fp2, "some data");
fprintf(fp2, "some more data");
fprintf(fp2, "even more data");

printf("Call div_zero...
");
fp(NULL);

printf("Error -- Should not get here\n");
exit(110);
}

Figure 166. The C program CELQSAMP (AMODE 64) (Part 2 of 2)

/* DLL containing div_zero */
#pragma options(SERVICE("1.4.f.0001"), NOOPT, GONUM)
#pragma export(div_zero)
#include <stdio.h>
#include <stdlib.h>
/* div_zero: Cause divide by zero exception */
******************************************************************************/
void *div_zero(void *parm)
{
    int i = 0;

    printf("Divide by zero...
");
i = 1/i;
    printf("Error -- Should not get here. i=%d\n",i);
    exit(110);
}

Figure 167. The C DLL CELQDLL (AMODE 64)

For easy reference, the sections of the following dump are numbered to correspond with the descriptions in "Sections of the Language Environment dump" on page 382.
Figure 168. Example dump using CEE3DMP (AMODE 64) (Part 1 of 9)
Figure 169. Example dump using CEE3DMP (AMODE 64) (Part 2 of 9)
Figure 170. Example dump using CEE3DMP (AMODE 64) (Part 3 of 9)
Figure 171. Example dump using CEE3DMP (AMODE 64) (Part 4 of 9)
<table>
<thead>
<tr>
<th>Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000001</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>0000000000000002</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>0000000000000003</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>0000000000000004</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>0000000000000005</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>0000000000000006</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>0000000000000007</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
<td>GP0R.....</td>
<td>0000000000000001</td>
</tr>
</tbody>
</table>

Control Blocks for Active Routines:

- **DSA** for **CEE3PLM2**: 000000000000001
- **DSA** for **CEEGPLM2**: 0000000000000001
- **DSA** for **CEEGPLCM**: 0000000000000001
- **DSA** for **CEEGPLCMM**: 0000000000000001

Figure 172. Example dump using CEE3DMP (AMODE 64) (Part 5 of 9)
Figure 173. Example dump using CEE3DMP (AMODE 64) (Part 6 of 9)
Figure 174. Example dump using CEE3DMP (AMODE 64) (Part 7 of 9)
Heap Storage Diagnostics
All storage has been freed.

Figure 175. Example dump using CEE3DMP (AMODE 64) (Part 8 of 9)
Figure 176. Example dump using CEE3DMP (AMODE 64) (Part 9 of 9)

Sections of the Language Environment dump
The sections of the dump listed in Table 53 appear independently of the Language Environment-conforming languages used.

Table 53. Contents of the Language Environment dump - AMODE 64

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Page Heading</td>
<td>The page heading section appears on the top of each page of the dump and contains:</td>
</tr>
<tr>
<td></td>
<td>• CEE3DMP identifier</td>
</tr>
<tr>
<td></td>
<td>• Title For dumps generated as a result of an unhandled condition, the title is “Condition processing resulted in the Unhandled condition.”</td>
</tr>
<tr>
<td></td>
<td>• Product abbreviation of Language Environment</td>
</tr>
<tr>
<td></td>
<td>• Version number</td>
</tr>
<tr>
<td></td>
<td>• Release number</td>
</tr>
<tr>
<td></td>
<td>• Date</td>
</tr>
<tr>
<td></td>
<td>• Time</td>
</tr>
<tr>
<td></td>
<td>• Page number</td>
</tr>
<tr>
<td>[2] CEE3845I CEE3DUMP Processing started.</td>
<td>Identifies the start of the Language Environment dump processing. Similarly, message CEE3845I can be used to locate the start of the next CEE3DUMP report when scanning forward in a data set that contains several CEE3DUMP reports.</td>
</tr>
</tbody>
</table>
Table 53. Contents of the Language Environment dump - AMODE 64 (continued)

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3] Enclave Identifier</td>
<td>Names the enclave for which information in the dump is provided.</td>
</tr>
<tr>
<td>[4] - [10] Thread Information:</td>
<td>These sections show information that is specific to a thread. When multiple threads are dumped, these sections will appear for each thread.</td>
</tr>
<tr>
<td>[4] Information for thread</td>
<td>Shows the system identifier for the thread. Each thread has a unique identifier.</td>
</tr>
<tr>
<td>[5] Traceback</td>
<td>For all active routines in a particular thread, the traceback section shows routine information in three parts. The first part contains:</td>
</tr>
<tr>
<td></td>
<td>• DSA number: A number that is assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback.</td>
</tr>
<tr>
<td></td>
<td>• Entry: For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, then the string ' NoName ' will appear.</td>
</tr>
<tr>
<td></td>
<td>• Entry point offset</td>
</tr>
<tr>
<td></td>
<td>• Statement number: Refers to the line number in the source code (program unit) in which a call was made or an exception took place. The statement number appears only if your routine was compiled with the options required to generate statement numbers. These options are described under &quot; XL C and XL C++ compiler options for AMODE 64 applications &quot; on page 351.</td>
</tr>
<tr>
<td></td>
<td>• Load module: The load module name displayed can be a partitioned data set member or an UNIX executable file. The load module name is also displayed in the third part of the traceback (see below for details).</td>
</tr>
<tr>
<td></td>
<td>• Program unit: The primary entry point of the external procedure. For C routines, this is the compile unit name. For Language Environment-conforming assemblers, this is the ENTNAME = value on the CELQPRLG macro. If your routine was compiled with the compile options to generate statement numbers then the program unit name displayed under this column will appear as follows:</td>
</tr>
<tr>
<td></td>
<td>– If your compiled routine is in a partitioned data set then only the member will be output.</td>
</tr>
<tr>
<td></td>
<td>– If your compiled routine is in a sequential data set then only the last qualifier will be shown.</td>
</tr>
<tr>
<td></td>
<td>– If your compiled routine is in an UNIX filename then only what fits of the filename will be displayed in a line.</td>
</tr>
<tr>
<td></td>
<td>Look for the complete name of the program unit in the Fully Qualified Names section of the traceback, if your routine was compiled using compile options to generate statement numbers.</td>
</tr>
<tr>
<td></td>
<td>• Service level: The latest service level applied to the compile unit (for example, for IBM products, it would be the PTF number).</td>
</tr>
<tr>
<td></td>
<td>– If the service level string is equal or less than 7 bytes, all of the string will be output.</td>
</tr>
<tr>
<td></td>
<td>– If the service level string is longer than 7 bytes, the Service column will only show the first 7 bytes of the service string, and the full service string will be shown in section of Full Service Level with max length of 64 bytes.</td>
</tr>
<tr>
<td></td>
<td>• Status: Routine status can be call or exception.</td>
</tr>
<tr>
<td>Section number and heading</td>
<td>Contents</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------</td>
</tr>
</tbody>
</table>
| [5] Traceback (continued)  | The second part contains:  
|                            | • DSA number: A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback. |
|                            | • Stack frame (DSA) address  
|                            | • Entry point address  
|                            | • Program unit address  
|                            | • Program unit offset: The offset of the last instruction to run in the routine. If the offset is a negative number, zero, or a very large positive number, the routine associated with the offset probably did not allocate a save area or the routine could have been called using SVC-assisted linkage. Adding the program unit address to the offset gives you the location of the current instruction in the routine. This offset is from the starting address of the routine.  
|                            | • Compile Date  
|                            | • Attributes: The attributes of the compile unit including whether character data is being treated as EBCDIC or ASCII and whether floating point data is being treated as IEEE or hexadecimal. |
|                            | The third part, which is also referred to as 'Fully Qualified Names' section, contains the following:  
|                            | • DSA number  
|                            | • Entry  
|                            | • Program unit: Similar to the Program Unit column in part 1 except that the server name and the complete program unit (PU) name will be displayed. A PU name will appear here only if it was compiled using compile options to produce statement numbers.  
|                            | • Load Module: The complete pathname of a load module name residing in an UNIX filename will be displayed here if available. The load module's full pathname will be displayed if the PATH environment variable is set such that the pathname of the load module's directory appears before the current directory (.). For load modules found in data sets, the same output shown in the traceback part 1 will also be displayed here. |
|                            | The fourth part of the traceback, which is also referred to as the “Full Service Level” section, contains the following:  
|                            | • DSA number  
|                            | • Entry  
<p>|                            | • Service: The full service level string with max length of 64 bytes will be displayed here. |</p>
<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [6] Condition Information for Active Routines | Displays the following information for all conditions currently active on the call chain:  
  • Statement showing failing routine and stack frame address of routine  
  • Condition information block (CIB) address  
  • The current condition, in the form of a Language Environment message for the condition raised or a Language Environment abend code, if the condition was caused by an abend  
  • Location: For the failing routine, this is the program unit, entry routine, statement number, and offset.  
  • Machine state, which shows:  
    – Instruction length counter (ILC)  
    – Interruption code  
    – Program status word (PSW)  
    – Contents of GPRs 0–15. Contents of floating point content register (FPC) and floating point registers FPR 0-15.  
    – Storage dump near condition (2 hex-bytes of storage near the PSW)  
    – Storage pointed to by General Purpose Registers  
  These values are the current values at the time the condition was raised. |
| [7] Parameters, Registers, and Variables for Active Routines | For each active routine, this section shows:  
  • Routine name and stack frame address  
  • Saved registers: This lists the contents of GPRs 0–15 at the time the routine received control. The saved registers are those saved by the DSA-owning routine on entry. Register 7 is the return address back to the caller of the DSA-owning routine. Register 6 may be the entry point of the DSA-owning routine. (This is not true when the Branch Relative and Save instruction is used to implement the call. The non-volatile floating-point registers that are saved in the stack frame. The registers are only displayed if the program owning the stack frame saved them. Dashes are displayed in the registers when the register values are not saved.  
  • Storage pointed to by the saved registers: Treating the saved contents of each register as an address, 32 bytes before and 64 bytes after the address shown. |
| [8] Control Blocks for Active Routines | For each active routine controlled by the STACKFRAME option, this section lists contents of related control blocks. The Language Environment-conforming language determines which language-specific control blocks appear. The possible control blocks are:  
  • Stack frame  
  • Condition information block  
  • Language-specific control blocks |
| [9] Storage for Active Routines | Displays local storage for each active routine. The storage is dumped in hexadecimal, with EBCDIC translations on the right side of the page. There can be other information, depending on the language used. For C/C++ routines, this is the stack frame storage. |
| [10] Control Blocks Associated with the Thread | Lists the contents of the Language Environment common anchor area (CAA), thread synchronization queue element (SQEL) and dummy stack frame. Other language-specific control blocks can appear in this section. |
Table 53. Contents of the Language Environment dump - AMODE 64 (continued)

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [11] Enclave Control Blocks | Lists the contents of the Language Environment enclave data block (EDB) and enclave member list (MEML). The information presented may vary depending on which runtime options are set.  
  • If the POSIX runtime option is set to ON, this section lists the contents of the mutex and condition variable control blocks, the enclave level latch table, and the thread synchronization trace block and trace table.  
  • If DLLs have been loaded, this section shows information for each DLL including the DLL name, load address, use count, writeable static area (WSA) address, and the thread ID of the thread that loaded the DLL.  
  • If the HEAPCHK runtime option is set to ON, this section shows the contents of the HEAPCHK options control block (HCOP) and the HEAPCHK element tables (HCEL). A HEAPCHK element table contains the location and length of all allocated storage elements for a heap in the order that they were allocated.  
  • When the call-level suboption of the HEAPCHK runtime option is set, any unfreed storage, which would indicate a storage leak, would be displayed in this area. The traceback could then be used to identify the program which did not free the storage.  
  • If the TRACE runtime option is set to ON, this section shows the contents of the Language Environment trace table.  
  Other language-specific control blocks can appear in this section. |
| [12] Runtime Options Report | Lists the Language Environment runtime options in effect when the routine was executed. |
| [13] Process Control Blocks | Lists the contents for the Language Environment process control block (PCB), process member list (MEML), and if the POSIX runtime option is set to ON, the process level latch table. Other language-specific control blocks can appear in this section. |
| [14] CEE3846I CEEDUMP | Identifies the end of the Language Environment dump processing. Similarly, message CEE3845I identifies the start of the dump processing. Message number CEE3846I can be used to locate the end of the previous CEEDUMP report when scanning backward in a data set that contains several CEEDUMP reports. |

Generating a system dump

A system dump contains the storage information needed to diagnose errors. You can use Language Environment to generate a system dump through any of the following methods:

**DYNDUMP**(hlq,DYNAMIC,TDUMP)  
You can use the DYNDUMP runtime option to obtain IPCS readable dumps of user applications that would ordinarily be lost due to the absence of a SYSDUMP, SYSUDUMP, or SYSABEND DD statement.

**TERMTHDACT**(UAONLY, UATRACE, or UADUMP)  
You can use these runtime options, with TRAP(ON), to generate a system dump if an unhandled condition of severity 2 or greater occurs. For further details regarding the level of dump information produced by each of the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 369.

**TRAP**(ON,NOSPIE) TERMTHDACT(UAIMM)  
TRAP(ON,NOSPIE) TERMTHDACT(UAIMM) generates a system dump of the user address space of the original abend or program interrupt prior to the Language Environment condition manager processing the condition.
Abend Codes in Initialization Assembler User Exit

Abend codes listed in the initialization assembler user exit are passed to the operating system. The operating system can then generate a system dump.

__cabend()

You can use the __cabend() API to cause the operating system to handle an abend.

See system or subsystem documentation for detailed system dump information.

The method for generating a system dump varies for each of the Language Environment runtime environments. The following sections describe the recommended steps needed to generate a system dump in batch and z/OS UNIX shell runtime environments. Other methods may exist, but these are the recommended steps for generating a system dump. For details on setting Language Environment runtime options, see z/OS Language Environment Programming Guide.

Steps for generating a system dump in a batch runtime environment

Perform the following steps to generate a system dump in a batch runtime environment. When you are done, you have a generated system dump in a batch runtime environment.

1. Specify runtime options TERMTHDACT(UAONLY, UADUMP, UATRACE, or UAIMM), and TRAP(ON). If you specify the suboption UAIMM then you must set TRAP(ON,NOSPIE). The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details on the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 369.

2. Decide whether to include a SYSMDUMP DD card or use the DYNDUMP runtime option.
   - Include a SYSMDUMP DD card with the desired data set name and DCB information:
     LRECL=4160, BLKSIZE=4160, and RECFM=FBS.
   - Specify the DYNDUMP runtime option with the following information:
     DYNDUMP (hlq,DYNAMIC,TDUMP)

3. Rerun the program.

Steps for generating a system dump in a z/OS UNIX shell

Perform the following steps to generate a system dump from a z/OS UNIX shell:

• Using _BPXK_MDUMP

1. Specify where to write the system dump.
   - To write the system dump to a z/OS data set, issue the export _BPXK_MDUMP=filename command, where filename is a fully qualified data set name with DCB information: LRECL=4160, BLKSIZE=4160, and RECFM=FBS.
     Example: export _BPXK_MDUMP=hlq.mydump
   - To write the system dump to an HFS file, issue the export _BPXK_MDUMP=filename command, where filename is a fully qualified HFS filename:
     Example: export _BPXK_MDUMP=/tmp/mydump.dmp
2. Specify Language Environment runtime options, where suboption = UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For more details regarding the TERMTHDACT suboptions, see "Generating a Language Environment dump with TERMTHDACT" on page 369.

   export _CEE_RUNOPTS="termthdact(suboption)"

3. Rerun the program.

   When you are done, the system dump is written to the data set name or HFS file name specified. For additional BPXK_MDUMP information, see z/OS UNIX System Services Command Reference.

   • Using DYNDUMP

   1. Specify Language Environment runtime options:

      export _CEE_RUNOPTS="termthdact(suboption),DYNDUMP(hlq,DYNAMIC,TDUMP)"

      suboption

      is UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details regarding the TERMTHDACT suboptions, see "Generating a Language Environment dump with TERMTHDACT" on page 369.

      hlq

      is the high level qualifier for the dump data set to be created.

   2. Rerun the program.

   When you are done, the system dump is written to the name generated by the DYNDUMP runtime option. For more DYNDUMP information see z/OS Language Environment Programming Reference.

   Note: You can also specify the signal SIGDUMP on the kill command to generate a system dump of the user address space. For more information about the SIGDUMP signal, see z/OS UNIX System Services Command Reference.

### Formatting and analyzing system dumps

You can use the Interactive Problem Control System (IPCS) to format and analyze system dumps. Language Environment provides an IPCS VERBEXIT LEDATA that can be used to format Language Environment control blocks. For more information on using IPCS, see z/OS MVS IPCS User’s Guide.

### Preparing to use the Language Environment support for IPCS

Use the following guidelines before you use IPCS to format Language Environment control blocks:

• Ensure that your IPCS job can find the CEEIPCSP member.

   IPCS provides an exit control table with imbed statements to enable other products to supply exit control information. The IPCS default table, BLSCECT, normally in the SYS1.PARMLIB library, has the following entry for Language Environment:

   IMBED MEMBER(CEEIPCSP) ENVIRONMENT(IPCS)

   The Language Environment-supplied CEEIPCSP member, installed in the SYS1.PARMLIB library, contains the Language Environment-specific entries for the IPCS exit control table.
• Provide an IPCSPARM DD statement to specify the libraries containing the IPCS control tables; for example:
//IPCSPARM DD DSN=SYS1.PARMLIB,DISP=SHR
• Ensure that your IPCS job can find the Language Environment-supplied ANALYZE exit routines installed in the SYS1.MIGLIB library.
• To aid in debugging system or address space hang situations, Language Environment mutexes, latches and condition variables can be displayed if the CEEIFCSP member you are using is updated to identify the Language Environment ANALYZE exit, by including the following statement:
EXIT EP(CEEEANLZ) ANALYZE

Understanding Language Environment IPCS VERBEXIT – LEDATA

Purpose

Use the LEDATA verb exit to format data for Language Environment. This VERBEXIT provides information about the following topics:
• A summary of Language Environment at the time of the dump
• Runtime Options
• Storage Management Control Blocks
• Condition Management Control Blocks
• Message Handler Control Blocks
• C/C++ Control Blocks
• PL/I Control Blocks

Format

<table>
<thead>
<tr>
<th>VERBEXIT LEDATA ['parameter[,parameter]...']</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report Type Parameters:</td>
</tr>
<tr>
<td>[ AUTH ]</td>
</tr>
<tr>
<td>[ NTHREADS(value) ]</td>
</tr>
<tr>
<td>[ SUM ]</td>
</tr>
<tr>
<td>[ HEAP</td>
</tr>
<tr>
<td>[ HPT(number)</td>
</tr>
<tr>
<td>[ CM ]</td>
</tr>
<tr>
<td>[ MH ]</td>
</tr>
<tr>
<td>[ CEEEDUMP ]</td>
</tr>
<tr>
<td>[ COMP(value) ]</td>
</tr>
<tr>
<td>[ PTBL(value) ]</td>
</tr>
<tr>
<td>[ ALL ]</td>
</tr>
<tr>
<td>Data Selection Parameters:</td>
</tr>
<tr>
<td>[ DETAIL</td>
</tr>
<tr>
<td>Control Block Selection Parameters:</td>
</tr>
<tr>
<td>[ CAA(caa-address) ]</td>
</tr>
<tr>
<td>[ DSA(dsa-address) ]</td>
</tr>
<tr>
<td>[ TCB(tcb-address) ]</td>
</tr>
<tr>
<td>[ ASID(address-space-id) ]</td>
</tr>
<tr>
<td>[ NTHREADS(value) ]</td>
</tr>
<tr>
<td>[ LAA(laa-address) ]</td>
</tr>
</tbody>
</table>

Chapter 12. Using Language Environment AMODE 64 debugging facilities 389
Parameters

The following sections describe the various types of parameters you can specify for VERBEXIT LEDATA. Only hexadecimal characters can be specified as addresses provided in LEDATA parameters. Special characters cause the formatter to fail. Therefore, to specify a 64 bit address as a parameter, it must be in the form like 123456789 instead of 1_23456789.

Report type parameters

Use these parameters to select the type of report. If you omit these parameters, the default is SUMMARY.

Address space report types: Use these parameters to select a report that shows the Language Environment activity for an address space. Only one of these reports may be specified.

**NTHREADS**(value)
Requests a report that shows the traceback for the TCBs in the address space. value is the number of TCBs for which the traceback will be displayed. If value is specified as asterisk (*), all TCBs will be displayed. The LAA, CAA, or TCB parameter can be used to limit the display to only TCBs that are part of the same enclave.

**AUTH**
Requests a report on all Preinitialized Environments for Authorized Programs control blocks for the address space. NTHREADS is ignored when AUTH is specified.

**PTBL**(value)
Requests that PreInit tables be formatted according to the following values.

**CURRENT**
If current is specified, the PreInit table that is associated with the current or specified TCB is displayed.

**address**
If an address is specified, the PreInit table at that address is specified.

**ACTIVE**
The PreInit tables for all TCBs in the address space are displayed.

Thread specific report types: Use these parameters to select reports that show Language Environment activity for a specific TCB. These report types are ignored if AUTH or NTHREADS is specified. You can specify as many of these reports as you wish.

**SUMmary**
Requests a summary of the Language Environment at the time of the dump.
The following information is included:
• TCB address
• Address Space Identifier
• Language Environment Release
• Active members
• Formatted CAA, PCB, RCB, EDB, LAA, and LCA
• Runtime Options in effect

HEAP | STACK | SM
HEAP
Requests a report on Storage Management control blocks pertaining to HEAP storage, as well as a detailed report on heap segments. The detailed report includes information about the free storage tree in the heap segment, and information about each allocated storage element. It also specifies a heap pools report with information useful to find potential damaged cells.

Note: Language Environment does not provide support for alternative Vendor Heap Manager (VHM) data.

STACK
Requests a report on Storage Management control blocks pertaining to STACK storage.

SM Requests a report on Storage Management control blocks. This is the same as specifying both HEAP and STACK.

\textbf{HPT}(\textit{number}) \textbf{[ HPTTCB }\textit{(address)} \textbf{]} \textbf{[ HPTCELL(\textit{address}) ] [ HPTLOC(\textit{location}) ]}

\textbf{HPT}(\textit{number})
Requests that the heap pool trace, if available, be formatted. If the value is 0 or *, the trace for every heap pool ID is formatted. If the value is a single number (1-12), the trace for the specific heap pool ID is formatted. If only the HPT keyword is specified with no value, the trace behaves similar to when the value is *. If no filter is specified, all of the entries are formatted for the specific pool ID.

\textbf{HPTTCB} (\textit{address})
Filters the heap pool trace table, if available, printing only those entries for a given TCB address (\textit{address}).

\textbf{HPTCELL}(\textit{address})
Filters the heap pool trace table, if available, printing only those entries for a given cell address (\textit{address}).

\textbf{HPTLOC}(\textit{value})
Filters the heap pool trace table, if available, and prints only those entries for a given virtual storage location (\textit{location}). The following values are valid:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Display entries that are located in virtual storage below the bar.</td>
</tr>
<tr>
<td>64</td>
<td>Display entries that are located in virtual storage above the bar.</td>
</tr>
<tr>
<td>ALL</td>
<td>Display entries that are located in virtual storage below or above the bar.</td>
</tr>
</tbody>
</table>

Note:
1. Filter options without specifying HPT implies HPT(*).
2. You can specify multiple options together, like HPTTCB and HPTCELL. All pieces of information must match the trace entry for it to be formatted. If location and cell contradict each other, such as HPTLOC(31) and HPTCELL(64bit addr), an error will be displayed.

\textbf{CM} Requests a report on Condition Management control blocks.

\textbf{MH} Requests a report on Message Handler control blocks.
**CEEDump**

Requests a CEEDUMP-like report. This includes the traceback, the Language Environment trace, and thread synchronization control blocks at process, enclave, and thread levels.

**COMP(value)**

Requests component control blocks to be formatted according to the following values:

- **C** Requests a report on C/C++ runtime control blocks.
- **CIO** Requests a report on C/C++ I/O control blocks.
- **COBOL** Requests a report on COBOL-specific control blocks.
- **PLI** Requests a report on PL/I-specific control blocks.
- **ALL** Requests a report on all the previous control blocks.

If the value specified in COMP is not one of the values (C, CIO, COBOL, PLI, or ALL), a message is displayed and it continues executing as if COMP(ALL) was specified.

The ALL parameter for LEDATA also generates a report that includes all the component control blocks.

**ALL**

Requests all the reports listed above, as well as C/C++, COBOL, and PL/I reports.

**Data selection parameters**

Data selection parameters limit the scope of the data in the report. If no data selection parameter is selected, the default is DETAIL.

- **DETail** Requests formatting all control blocks for the selected components. Only significant fields in each control block are formatted. For the Heap and Storage Management Reports, the DETAIL parameter will provide a detailed heap segment report for each heap segment in the dump. The detailed heap segment report includes information on the free storage tree in the heap segments, and all allocated storage elements. This report will also identify problems detected in the heap management data structures. For more information about the Heap Reports, see "Understanding the HEAP LEDATA output" on page 408.

- **EXCEPTION** Requests validating all control blocks for the selected components. Output is only produced naming the control block and its address for the first control block in a chain that is invalid. Validation consists of control block header verification at the very least. For the Summary, CEEDUMP, C/C++, PL/I reports, the EXCEPTION parameter has not been implemented. For these reports, DETAIL output is always produced.

**Control block selection parameters**

Use these parameters to select the control blocks used as the starting points for formatting.
CAA\((\text{caa-address})\)
specifies the address of the CAA. If not specified, the CAA address is obtained from the LAA.

DSA\((\text{dsa-address})\)
specifies the address of the DSA. If not specified, the DSA address may be obtained from the TCB or the IPCS symbol REGGEN.

TCB\((\text{tcb-address})\)
specifies the address of the TCB. If not specified, the TCB address may be obtained from the CAA or the CVT.

LAA\((\text{laa-address})\)
specifies the address of the LAA. If not specified, the LAA address may be obtained from the TCB or the PSA.

ASID\((\text{address-space-id})\)
specifies the hexadecimal address space ID. If not specified, the IPCS default address space ID is used. This parameter is not needed when the dump only has one address space.

Examples
For examples of the output produced by LEDATA and explanation of the content, refer to “Understanding the Language Environment IPCS VERBEXIT LEDATA output.”

Understanding the Language Environment IPCS VERBEXIT LEDATA output
The Language Environment IPCS VERBEXIT LEDATA generates formatted output of the Language Environment runtime environment control blocks from a system dump. The following sample illustrates the output produced when the LEDATA VERBEXIT is invoked with the ALL parameter. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option when running the program CELQSAMP in Figure 165 on page 372.

“Sections of the Language Environment LEDATA VERBEXIT formatted output” on page 404 describes the information in the formatted output. Ellipses are used to summarize some sections of the dump. For easy reference, the sections of the following dump are numbered to correspond with the descriptions in “Sections of the Language Environment LEDATA VERBEXIT formatted output” on page 404.
Figure 177. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 10)
7

Figure 178. Example of formatted output from LEDA VERBEXIT (AMODE 64) (Part 2 of 10)
Figure 179. Example of formatted output from LEDA VERBEXIT (AMODE 64) (Part 3 of 10)
Figure 180. Example of formatted output from LEDA VERBEXIT (AMODE 64) (Part 4 of 10)
Figure 181. Example of formatted output from LEDA VERBEXIT (AMODE 64) (Part 5 of 10)
Figure 182. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 6 of 10)
Figure 183. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 7 of 10)
Figure 184. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 8 of 10)
Figure 185. Example of formatted output from LEDA VERBEXIT (AMODE 64) (Part 9 of 10)
Chapter 12. Using Language Environment AMODE 64 debugging facilities

Figure 186. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 10 of 10)
Sections of the Language Environment LEDATA VERBEXIT formatted output

Table 54 lists the sections of the LEDATA VERBEXIT output, which appear independently of the Language Environment-conforming languages used.

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] - [8] CEEDUMP Formatted Control Blocks</td>
<td>These sections are included when the CEEDUMP parameter is specified on the LEDATA invocation.</td>
</tr>
<tr>
<td>[1] - [4] NTHREADS data</td>
<td>These sections are also included, once for each thread, when the NTHREADS() parameter is specified on the LEDATA invocations. For a description of NTHREADS, see Report type parameters.</td>
</tr>
<tr>
<td>[1] Enclave Identifier</td>
<td>Names the enclave for which information is provided.</td>
</tr>
<tr>
<td>[2] Information for thread</td>
<td>Shows the system identifier for the thread. Each thread has a unique identifier.</td>
</tr>
<tr>
<td>[3] Registers and PSW</td>
<td>Displays the register and program status word (PSW) values that were used to create the traceback. These values may come from the TCB, the RTM2 work area, a linkage stack entry or output from the BPXGMSTA service. This section is not displayed when the DSA() parameter is specified on the LEDATA invocation.</td>
</tr>
</tbody>
</table>
### Table 54. Contents of the Language Environment LEDA VERBEXIT formatted output (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4] Traceback</td>
<td>For all active routines in a particular thread, the traceback section shows routine information in two parts. The first part contains:</td>
</tr>
<tr>
<td></td>
<td>• DSA number: A number assigned to the information for this active routine by LEDATA. The number is only used to associate information from the first part of the traceback with information in the second part of the traceback.</td>
</tr>
<tr>
<td></td>
<td>• Entry: For PL/I routines, this is the entry point name. For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, then the string &quot;** NoName **&quot; will appear.</td>
</tr>
<tr>
<td></td>
<td>• Entry point offset</td>
</tr>
<tr>
<td></td>
<td>• Load module</td>
</tr>
<tr>
<td></td>
<td>• Program unit: The primary entry point of the external procedure. For C and PL/I routines, this is the compile unit name. For Language Environment-conforming assemblers, this is the ENTNAME = value on the CELQPR LG macro.</td>
</tr>
<tr>
<td></td>
<td>• Service level: The latest service level applied to the compile unit (for example, for IBM products, it would be the PTF number.</td>
</tr>
<tr>
<td></td>
<td>– If the service level string is equal or less than 7 bytes, all of the string will be output.</td>
</tr>
<tr>
<td></td>
<td>– If the service level string is longer than 7 bytes, the Service column will only show the first 7 bytes of the service string, and the full service string will be shown in section of Full Service Level with max length of 64 bytes.</td>
</tr>
<tr>
<td></td>
<td>• Status: Routine status can be call, exception, or running.</td>
</tr>
<tr>
<td></td>
<td>The second part contains:</td>
</tr>
<tr>
<td></td>
<td>• DSA number: A number assigned to the information for this active routine by LEDATA. The number is only used to associate information from the first part of the traceback with information in the second part of the traceback.</td>
</tr>
<tr>
<td></td>
<td>• Stack frame (DSA) address</td>
</tr>
<tr>
<td></td>
<td>• Entry point address</td>
</tr>
<tr>
<td></td>
<td>• Program unit address</td>
</tr>
<tr>
<td></td>
<td>• Program unit offset: The offset of the last instruction to run in the routine. If the offset is a negative number, zero, or a very large positive number, the routine associated with the offset probably did not allocate a save area, or the routine could have been called using SVC-assisted linkage. Adding the program unit address to the offset gives you the location of the current instruction in the routine. This offset is from the starting address of the routine.</td>
</tr>
<tr>
<td></td>
<td>The third part of the traceback, which is also referred to as the “Full Service Level” section, contains the following:</td>
</tr>
<tr>
<td></td>
<td>• DSA number</td>
</tr>
<tr>
<td></td>
<td>• Entry</td>
</tr>
<tr>
<td></td>
<td>• Service: The full service level string with max length of 64 bytes will be displayed here.</td>
</tr>
<tr>
<td>[5] Control Blocks Associated with the Thread</td>
<td>Lists the contents of the thread synchronization queue element (SQEL).</td>
</tr>
<tr>
<td>[6] Enclave Control Blocks</td>
<td>If the POSIX runtime option was set to ON, this section lists the contents of the mutex and condition variable control blocks, the enclave level latch table, and the thread synchronization trace block and trace table. If the HEAPCHK runtime option is set to ON, this section lists the contents of the HEAPCHK options control block (HCOP) and the HEAPCHK element tables (HCEL). A HEAPCHK element table contains the location and length of all allocated storage elements for a heap in the order that they were allocated.</td>
</tr>
<tr>
<td>Section Number and Heading</td>
<td>Contents</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>[7] Language Environment Trace Table</td>
<td>If the TRACE runtime option was set to ON, this section shows the contents of the Language Environment trace table.</td>
</tr>
<tr>
<td>[8] Process Control Blocks</td>
<td>If the POSIX runtime option was set to ON, this section lists the contents of the process level latch table.</td>
</tr>
<tr>
<td>[9] - [17] Summary: These sections are included when the SUMMARY parameter is specified on the LEDATA invocation.</td>
<td></td>
</tr>
<tr>
<td>[9] Summary Header</td>
<td>The summary header section contains:</td>
</tr>
<tr>
<td></td>
<td>• Address of Thread control block (TCB)</td>
</tr>
<tr>
<td></td>
<td>• Release number</td>
</tr>
<tr>
<td></td>
<td>• Address Space ID (ASID)</td>
</tr>
<tr>
<td>[10] Active Members List</td>
<td>Lists active members, which is extracted from the enclave member list (MEML).</td>
</tr>
<tr>
<td>[12] CEELCA</td>
<td>Formats the contents of the Language Environment library control area (LCA). See z/OS Language Environment Vendor Interfaces for a description of the fields in the LCA.</td>
</tr>
<tr>
<td>[13] CEECAA</td>
<td>Formats the contents of the Language Environment common anchor area (CAA). See z/OS Language Environment Vendor Interfaces for a description of the fields in the CAA. If there is any, DLL failure data is also formatted.</td>
</tr>
<tr>
<td>[14] CEEPCB</td>
<td>Formats the contents of the Language Environment process control block (PCB), and the process level member list.</td>
</tr>
<tr>
<td>[16] CEEEDB</td>
<td>Formats the contents of the Language Environment enclave data block (EDB), and the enclave level member list.</td>
</tr>
<tr>
<td>[17] Runtime Options</td>
<td>Lists the runtime options in effect at the time of the dump, and indicates where they were set.</td>
</tr>
<tr>
<td>[18] Heap Storage Control Blocks</td>
<td>This section is included when the HEAP or SM parameter is specified on the LEDATA invocation. It formats the Enclave-level storage management control block (ENSQ) and for each different type of heap storage:</td>
</tr>
<tr>
<td></td>
<td>• Heap control block (HPCQ)</td>
</tr>
<tr>
<td></td>
<td>• Chain of heap anchor blocks (HANQ). A HANQ immediately precedes each segment of heap storage.</td>
</tr>
<tr>
<td></td>
<td>This section includes a detailed heap segment report for each segment in the dump. For more information about the detailed heap segment report, see “Understanding the HEAP LEDATA output” on page 408.</td>
</tr>
<tr>
<td>[19] Stack Storage Control Blocks</td>
<td>This section is included when the STACK or SM parameter is specified on the LEDATA invocation. This section formats:</td>
</tr>
<tr>
<td></td>
<td>• Stack anchor (SANC)</td>
</tr>
<tr>
<td></td>
<td>• Chain of dynamic save areas (DSA)</td>
</tr>
<tr>
<td>[20] Condition Management Control Blocks</td>
<td>This section is included when the CM parameter is specified on the LEDATA invocation. It formats the chain of Condition Information Block Headers (CIBH) and Condition Information Blocks. The Machine State Information Block is contained with the CIBH starting with the field labeled MCH_EYE.</td>
</tr>
<tr>
<td>[21] Message Processing Control Blocks</td>
<td>This section is included when the MH parameter is specified on the LEDATA invocation.</td>
</tr>
</tbody>
</table>
PTBL LEDATA output: The Language Environment IPCS VERBEXIT LEDATA command generates formatted output of PreInit tables when the PTBL or ALL parameter are specified. If ALL is specified, PTBL defaults to CURRENT value. Figure 187 illustrates the output produced when the VERBEXIT LEDATA command is invoked with the PTBL parameter.

**Figure 187. Example of formatted PTBL output from LEDATA VERBEXIT (Part 1 of 2)**
Understanding the HEAP LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap segment report when the HEAP option is used with the DETAIL option, or when the SM, DETAIL option is specified. The detailed heap segment report is useful when trying to pinpoint damage because it provides specific information. The report describes the nature of the damage, and specifies where the actual damage occurred. The report can also be used to diagnose storage leaks, and to identify heap fragmentation. "Heap report sections of the LEDATA output" on page 416 describes the information in the formatted output.

For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows. Ellipses are used to summarize some sections of the dump.

Note: Language Environment does not provide support for alternative Vendor Heap Manager (VHM) data. LEDATA VERBEXIT will state that an alternative VHM is in use.

Figure 188. Example of formatted PTBL output from LEDATA VERBEXIT (Part 2 of 2)
Figure 189. Example formatted detailed heap segment report from LEDA VERBXEXIT (AMODE 64) (Part 1 of 8)
Figure 190. Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 2 of 8)
**Figure 191. Example formatted detailed heap segment report from LEDA VERBEXIT (AMODE 64) (Part 3 of 8)**
Figure 192. Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 4 of 8)
This is the last heap segment in the current heap.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Node</th>
<th>Length</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25774168</td>
<td>0</td>
<td>00002E98</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Summary of analysis for Heap Segment 25773000:
Amounts of identified storage:
Free: 00002E98
Allocated: 00001128
Total: 00003FC0
Number of identified areas:
Free: 1
Allocated: 1
Total: 2
00000000 bytes of storage were not accounted for.
No errors were found while processing this heap segment.
This is the last heap segment in the current heap.

User Heap Control Blocks

** No segments allocated **

Library Heap Control Blocks

** No segments allocated **

Figure 193. Example formatted detailed heap segment report from LEDA VERBEXIT (AMODE 64) (Part 5 of 8)
Figure 194. Example formatted detailed heap segment report from LEDA VERBEXIT (AMODE 64) (Part 6 of 8)
Figure 195. Example formatted detailed heap segment report from LEDA VERBEXIT (AMODE 64) (Part 7 of 8)
Figure 196. Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 8 of 8)

Heap report sections of the LEDATA output
The Heap Report sections of the LEDATA output provide information for each heap segment in the dump. The detailed heap segment reports include information on the free storage tree in the heap segments, the allocated storage elements, and the cause of heap management data structure problems.
Table 55. Contents of Heap report sections of the LEDATA output (AMODE 64)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [1] Free Storage Tree Report   | Within each heap segment, Language Environment keeps track of unallocated storage areas by chaining them together into a tree. Each free area represents a node in the tree. Each node contains a header, which points to its left and right child nodes. The header also contains the length of each child. The LEDATA HEAP option formats the free storage tree within each heap, and validates all node addresses and lengths within each node. Each node address is validated to ensure that it:  
  • Falls on a doubleword boundary  
  • Falls within the current heap segment  
  • Does not point to itself  
  • Does not point to a node that was previously traversed  
  Each node length is validated to ensure that it:  
  • Is a multiple of 8  
  • Is not larger than the heap segment length  
  • Does not cause the end of the node to fall outside of the current heap segment  
  • Does not cause the node to overlap another node  
  If the formatter finds a problem, then it will place an error message describing the problem directly after the formatted line of the node that failed validation. |
| [2] Heap Segment Map Report     | The LEDATA HEAP option produces a report that lists all of the storage areas within each heap segment, and identifies the area as either allocated or freed. For each allocated area the contents of the first X'20' bytes of the area are displayed in order to help identify the reason for the storage allocation. Each allocated storage element has a prefix used by Language Environment to manage the area. The prefix contains a pointer to the start of the heap segment followed by the length of the allocated storage element. For HEAP64 heaps, the prefix is 16 bytes, with 8-byte pointer and length fields. For HEAP31 and HEAP24 heaps, the pointer is 8 bytes with 4-byte pointer and length field. The formatter validates this header to ensure that its heap segment pointer is valid. The length is also validated to ensure that it:  
  • Is a multiple of 8  
  • Is not zero  
  • Is not larger than the heap segment length  
  • Does not cause the end of the element to fall outside of the current heap segment  
  • Does not cause the element to overlap a free storage node  
  If the heap_free_value of the STORAGE runtime option was specified, then the formatter also checks that the free storage within each free storage element is set to the requested heap_free_value. If a problem is found, then an error message describing the problem is placed after the formatted line of the storage element that failed validation. |

Diagnosing heap damage problems

Heap storage errors can occur when an application allocates a heap storage element that is too small for it to use, and therefore, accidently overlays heap storage. If this situation occurs then some of the typical error messages generated are:
  • The node address does not represent a valid node within the heap segment  
  • The length of the segment is not valid, or  
  • The heap segment pointer is not valid.

If one of the above error messages is generated by one of the reports, then examine the storage element that immediately precedes the damaged node to determine if this storage element is owned by the application program. Check the size of the storage element and ensure that it is sufficient for the program's use. If the size of the storage element is not sufficient then adjust the allocation size.
If an error occurs indicating that the node’s pointers form a circular loop within the free storage tree, then check the Free Storage Tree Report to see if such a loop exists. If a loop exists, then contact the IBM support center for assistance because this may be a problem in the Language Environment heap management routines.

Additional diagnostic information regarding heap damage can be obtained by using the HEAPCHK runtime option. This option provides a more accurate time perspective on when the heap damage actually occurred, which could help to determine the program that caused the damage. For more information on HEAPCHK, see z/OS Language Environment Programming Reference.

**Diagnosing storage leak problems**
A storage leak occurs when a program does not return storage back to the heap after it has finished using it. To determine if this problem exists, do one of the following:

- The call-level suboption of the HEAPCHK runtime option causes a report to be produced in the CEEDUMP. Any still-allocated (that is, not freed) storage identified by HEAPCHK is listed in the report, along with the corresponding traceback. This shows any storage that wasn’t freed, as well as all the calls that were involved in allocating the storage. For more information about the HEAPCHK runtime option, see z/OS Language Environment Programming Reference.

- Examine the Heap Segment Map report to see if any data areas, within the allocated storage elements, appear more frequently than expected. If they do, then check to see if these data areas are still being used by the application program. If the data areas are not being used, then change the program to free the storage element after it is done with it.

**Diagnosing heap fragmentation problems**
Heap fragmentation occurs when allocated storage is interlaced with many free storage areas that are too small for the application to use. Heap fragmentation could indicate that the application is not making efficient use of its heap storage. Check the Heap Segment Map report for frequent free storage elements that are interspersed with the allocated storage elements.

**Understanding the heap pool LEDATA output**
The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap pool report when HEAPPOOLS is ON. The detailed heap pool report is useful when trying to find potential damaged cells because it provides very specific information. Figure 197 on page 419 illustrates the details of the heap pool report. “Heap pool report sections of the LEDATA output” on page 423 describes the information contained in the formatted output.
Heap Pool Report

Data for pool 1:

EXTENT: 00000008_0862E6B0

To display entire pool extent: IP LIST 00000008_0862E6B0 LEN(X'00000000') ASID(X'0021')

To display entire pool extent: IP LIST 00000008_0862E650 LEN(X'00000000') ASID(X'0021')

Summary of analysis for Pool 1:
Number of cells: Unused: 9 Free: 1 Allocated: 0 Total Used: 10
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 2:

EXTENT: 00000008_0862E650

To display entire pool extent: IP LIST 00000008_0862E650 LEN(X'00000000') ASID(X'0021')

To display entire pool extent: IP LIST 00000008_0862E660 LEN(X'00000000') ASID(X'0021')

Summary of analysis for Pool 2:
Number of cells: Unused: 3 Free: 1 Allocated: 0 Total Used: 4
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 3:

EXTENT: 00000008_0862E680

To display entire pool extent: IP LIST 00000008_0862E680 LEN(X'00000000') ASID(X'0021')

To display entire pool extent: IP LIST 00000008_0862E690 LEN(X'00000000') ASID(X'0021')

Summary of analysis for Pool 3:
Number of cells: Unused: 3 Free: 1 Allocated: 0 Total Used: 4
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Figure 197. Example formatted detailed heap pool report from LEDA VERBEXIT (AMODE 64) (Part 1 of 5)
Heap Pool Extent Mapping

EXTENT: 00000000_0862E790
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:0000000B_0862E2F0
To display entire pool extent: IP LIST 00000000B0862E790 LEN(X'000000250') ASID(X'0021')
00000000B08627A0: Free storage cell. To display: IP LIST 00000000B08627A0 LEN(X'000000090') ASID(X'0021')
00000000B0862E30: Free storage cell. To display: IP LIST 00000000B0862E30 LEN(X'000000090') ASID(X'0021')
00000000B0862E3A0: Allocated storage cell. To display: IP LIST 00000000B0862E3A0 LEN(X'000000090') ASID(X'0021')
00000000B0862E8B0: Allocated storage cell. To display: IP LIST 00000000B0862E8B0 LEN(X'000000090') ASID(X'0021')
00000000B0862E8B0: Data

EXTENT: 00000000_0862E2F0
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_0862E2F0
To display entire pool extent: IP LIST 00000000B0862E2F0 LEN(X'000000250') ASID(X'0021')

Verification free chain for pool: 3...
No errors were found while processing free chain.

Summary of analysis for Pool 3:
Number of cells: Unused: 1 Free: 2 Allocated: 5 Total Used: 8
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 4:
POOLDATA: 00000000_08734000
+000000 POOL_INDEX:00000004 INPUT_CELL_SIZE:00000100
+000000 CELL_POOL_SIZE:00000110 INPUT_COUNT:00000004
+000010 CELL_POOL_SIZE:00000440 CELL_POOL_NUM:0000000004
+000018 POOL_LATCH_ADDR:00000000_08711588 POOL_EXTENTS:00000001
+000028 LAST_CELL:00000008_0862E3D0 NEXT_CELL:00000008_0862E2D0
+000040 Q_CONTROL_INFO:00000000 00000000 00000000 00000000
+000050 POOL_NUM_GET_TOTAL:00000000 00000000
+000058 POOL_NUM_FREE:00000000 00000000 POOL_EXTENTS_ANCHOR:00000000_0862E9F0
+000068 POOL_INDEXSAME_SIZE:01 POOL_INDEX_SIZE:04
+00006A POOL_INDEXSAME_SIZE:01 POOL_TRACE_TABLE:00000000_088D2170

Heap Pool Extent Mapping

EXTENT: 00000000_0862E9F0
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_0862E9F0
To display entire pool extent: IP LIST 00000000B0862E9F0 LEN(X'000000250') ASID(X'0021')
00000000B0862E30: Allocated storage cell. To display: IP LIST 00000000B0862E30 LEN(X'000000090') ASID(X'0021')
00000000B0862E40: Allocated storage cell. To display: IP LIST 00000000B0862E40 LEN(X'000000090') ASID(X'0021')
00000000B0862E8B0: Allocated storage cell. To display: IP LIST 00000000B0862E8B0 LEN(X'000000090') ASID(X'0021')
00000000B0862E8B0: Data

Verification free chain for pool: 4...
No errors were found while processing free chain.

Summary of analysis for Pool 4:
Number of cells: Unused: 2 Free: 2 Allocated: 0 Total Used: 4
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 5.1:
POOLDATA: 00000000_08734100
+000000 POOL_INDEX:00000004 INPUT_CELL_SIZE:00000400
+000000 CELL_POOL_SIZE:00000410 INPUT_COUNT:00000004
+000010 CELL_POOL_SIZE:000002000 CELL_POOL_NUM:0000000004
+000018 POOL_LATCH_ADDR:00000000_08711588 POOL_EXTENTS:00000001
+000028 LAST_CELL:00000008_08605B10 NEXT_CELL:00000008_08605B30
+000040 Q_CONTROL_INFO:00000000 00000000 00000000 00000000
+000050 POOL_NUM_GET_TOTAL:00000000 00000000
+000058 POOL_NUM_FREE:00000000 00000000 POOL_EXTENTS_ANCHOR:00000000_0862E9F0
+000068 POOL_INDEXSAME_SIZE:01 POOL_INDEX_SIZE:04
+00006A POOL_INDEXSAME_SIZE:01 POOL_TRACE_TABLE:00000000_088D2170

Figure 198. Example formatted detailed heap pool report from LEDA VERBEXIT (AMODE 64) (Part 2 of 5)
Figure 199. Example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64) (Part 3 of 5)
Data for pool 5:

POOLDATA: 00000008_08734500
+000000 POOL_INDEX:00000009  INPUT_CELL_SIZE:00000400
+000000 CELL_SIZE:00000040  INPUT_COUNT:00000004
+000010 CELL_POOL_SIZE:00001040  CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:00000008_08711800  POOL_EXTENTS:00000000
+000020 LAST_CELL:00000000_00000000  NEXTCELL:00000000_00000000
+000040 Q_CONTROL_INFO:00000000  Q_FIRSTCELL:00000000_00000000
+000050 POOL_NUM_GET_TOTAL:00000000  POOL_TRACE_TABLE:00000000_08A30260
+00006A POOL_NUMSAME_SIZE:05  POOL_INDEX:00000005

There are no extents for this pool.

Data for pool 6:

POOLDATA: 00000008_08734600
+000000 POOL_INDEX:0000000A  INPUT_CELL_SIZE:00000000
+000000 CELL_SIZE:00000000  INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00002040  CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:00000008_087118A0  POOL_EXTENTS:00000001
+000020 LAST_CELL:00000000_00000000  NEXT_CELL:00000000_00000000
+000040 Q_CONTROL_INFO:00000000  Q_FIRST_CELL:00000000_00000000
+000050 POOL_NUM_GET_TOTAL:00000000  POOL_TRACE_TABLE:00000000_08A30260
+00006A POOL_NUMSAME_SIZE:01  POOL_INDEX:00000001

EXTENT: 00000008_0862C290
+000000 EYE_CATCHER:EX64  NEXT_EXTENT:00000000_00000000
000000000862C2A0: Allocated storage cell. To display: IP LIST 000000000862C2A0 LEN('X'00000810') ASID('X'0021')
000000000862C2B0: Free storage cell. To display: IP LIST 000000000862C2B0 LEN('X'00000810') ASID('X'0021')
000000000862C2C0: Allocated storage cell. To display: IP LIST 000000000862C2C0 LEN('X'00000810') ASID('X'0021')
000000000862C2D0: Free storage cell. To display: IP LIST 000000000862C2D0 LEN('X'00000810') ASID('X'0021')
000000000862C2E0: Free storage cell. To display: IP LIST 000000000862C2E0 LEN('X'00000810') ASID('X'0021')
000000000862C2F0: Free storage cell. To display: IP LIST 000000000862C2F0 LEN('X'00000810') ASID('X'0021')

[1] Verifying free chain for pool: 6...

No errors were found while processing free chain.

Summary of analysis for Pool 6:

Number of cells: Unused: 1 Free: 1 Allocated: 2 Total Used: 4
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 7:

POOLDATA: 00000008_08734700
+000000 POOL_INDEX:0000000B  INPUT_CELL_SIZE:00000000
+000000 CELL_SIZE:00000000  INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00002580  CELL_POOL_NUM:00000032
+000018 POOL_LATCH_ADDR:00000008_087118B0  POOL_EXTENTS:00000001
+000020 LAST_CELL:00000000_00000000  NEXT_CELL:00000000_00000000
+000040 Q_CONTROL_INFO:00000000  Q_FIRST_CELL:00000000_00000000
+000050 POOL_NUM_GET_TOTAL:00000000  POOL_TRACE_TABLE:00000000_08A30260
+00006A POOL_NUMSAME_SIZE:01  POOL_INDEX:00000001

EXTENT: 00000008_08606750
+000000 EYE_CATCHER:EX64  NEXT_EXTENT:00000000_00000000
To display entire pool extent: IP LIST 0000000008606750 LEN('X'00000810') ASID('X'0021')
0000000008606760: Allocated storage cell. To display: IP LIST 0000000008606760 LEN('X'00000810') ASID('X'0021')
0000000008606770: Free storage cell. To display: IP LIST 0000000008606770 LEN('X'00000810') ASID('X'0021')
0000000008606780: Free storage cell. To display: IP LIST 0000000008606780 LEN('X'00000810') ASID('X'0021')

Summary of analysis for Pool 7:

Number of cells: Unused: 48 Free: 0 Allocated: 2 Total Used: 50
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 8:

POOLDATA: 00000008_08734800
+000000 POOL_INDEX:0000000C  INPUT_CELL_SIZE:00000000
+000000 CELL_SIZE:00000000  INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00003232  CELL_POOL_NUM:00000032
+000018 POOL_LATCH_ADDR:00000008_087118F8  POOL_EXTENTS:00000000

Summary of analysis for Pool 8:

No errors were found while processing this Pool.

Figure 200. Example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64) (Part 4 of 5)
Heap pool report sections of the LEDATA output

As Table 56 on page 424 shows, the heap pool report provides information about the following items:

- Each cell pool.
- The free chain associated with every qpcb pool data area, and all the free and allocated cells in the extent chain.
- Errors found when the cells are validated.
### Table 56. Contents of the heap pool report sections of the LEDA output (AMODE 64)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Free Chain Validation</td>
<td>Within each cell pool, Language Environment keeps track of unallocated cells by chaining them together. The LEDA HEAP option validates the free chain within each cell pool. It verifies that the cell pointer is within a valid extent and that the cell pool number is valid. If the formatter finds a problem, it will place an error message describing the problem directly after the formatted line of the cell that failed validation.</td>
</tr>
<tr>
<td>[2] Heap Pool Extent Mapping Report</td>
<td>The LEDA HEAP option produces a report that lists all of the cells within each pool extent, and identifies the cells as either allocated or free. For each allocated cell, the contents of the first X'20' bytes of the area are displayed to identify the reason for the storage allocation. The formatter validates if the cell pool number in header is correct.</td>
</tr>
</tbody>
</table>

### Understanding the heap pools trace LEDA output

The Language Environment IPCS VERBEXIT LEDA generates a detailed heap pools trace report when the HPT option is used (see Figure 202). The argument value is the ID of the pool to be formatted in the report. Table 57 on page 428 explains the contents of each section of the report.

```
HPT(3)
*****************************************************************************
64 BIT LANGUAGE ENVIRONMENT DATA
*****************************************************************************

Language Environment Product 04 V01 R0A.00

[1] HEAPPOOLS64 Trace Table

  Type: FREE  Cell Address: 00000001086588E0 Cpid: 01  Tcb: 008D7820

  CALL NAME       CALL ADDRESS       CALL OFFSET
  GetStorage::~GetStorage() 000000025B000180 000000056
  foo8()          000000025B00348  000000046
  foo7()          000000025B00348  000000010
  foo6()          000000025B00348  000000010
  foo5()          000000025B00348  000000010
  foo4()          000000025B00348  000000010
  foo3()          000000025B00348  000000010
  foo2()          000000025B00348  000000010
  foo1()          000000025B00348  000000010
  thread         000000025B00348  000000000

  Type: FREE  Cell Address: 0000000108658970 Cpid: 01  Tcb: 008D7820

  CALL NAME       CALL ADDRESS       CALL OFFSET
  GetStorage::~GetStorage() 000000025B000180 000000056
  foo9()          000000025B00260  00000006E
  foo8()          000000025B00348  000000046
  foo7()          000000025B00348  000000010
  foo6()          000000025B00348  000000010
  foo5()          000000025B00348  000000010
  foo4()          000000025B00348  000000010
  foo3()          000000025B00348  000000010
  foo2()          000000025B00348  000000010
  foo1()          000000025B00348  000000010
```

Figure 202. Example of formatted detailed heap pools trace report from LEDA VERBEXIT (AMODE 64) (Part 1 of 5)
Figure 203. Example of formatted detailed heap pools trace report from LEDA VERBEXIT (AMODE 64) (Part 2 of 5)
Figure 204. Example of formatted detailed heap pools trace report from LEDA VERBEXIT (AMODE 64) (Part 3 of 5)
Figure 205. Example of formatted detailed heap pools trace report from LEDA VERBEXIT (AMODE 64) (Part 4 of 5)
Table 57. Contents of a detailed heap pools trace report from LEDA VERBEXIT (AMODE 64) (Part 5 of 5)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Trace Header</td>
<td>HEAPPOOLS64 trace header information.</td>
</tr>
<tr>
<td>[2] Pool Information</td>
<td>Includes the number of the pool (pool ID) that is currently being formatted, the ASID, the number of entries formatted, and the total number of entries taken. The trace wraps for each pool ID after a specific number of entries. The number of entries is controlled by the HEAPCHK runtime option.</td>
</tr>
<tr>
<td>[3] Timestamp</td>
<td>The time this trace entry was taken. The trace entries are formatted in reverse order (most recent trace entry first).</td>
</tr>
</tbody>
</table>
| [4] Trace Table Entry contents | The individual trace entry, which contains:  
  • The TYPE - GET or FREE.  
  • The Cell within the pool being acted upon.  
  • The CPU and TCB that requested or freed the cell.  
  • A traceback at the time of the request. The number of entries in this traceback is limited by the HEAPCHK runtime option. |

Understanding the C/C++-specific LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of C/C++-specific control blocks from a system dump when the COMP(C), COMP(ALL), or ALL parameter is specified and C/C++ is active in the dump. Figure 207 on page 429 illustrates the C/C++-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option when running the program CELOSAMP. Figure 165 on page 372 “C/C++-specific sections of the LEDATA output” on page 438 describes the information contained in the formatted output. Ellipses are used to summarize some sections of the dump. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.
Figure 207. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 10)
Figure 208. Example of formatted C/C++ output from LEDA VERBEXIT (AMODE 64) (Part 2 of 10)
Figure 209. Example of formatted C/C++ output from LEDA VERBEXIT (AMODE 64) (Part 3 of 10)
Figure 210. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 4 of 10)
Figure 211. Example of formatted C/C++ output from LEDA VERBEXIT (AMODE 64) (Part 5 of 10)
Figure 212. Example of formatted C/C++ output from LEDA VERBEXIT (AMODE 64) (Part 6 of 10)
Figure 213. Example of formatted C/C++ output from LEDA VERBEXIT (AMODE 64) (Part 7 of 10)
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCB:</td>
<td>00000000_00007E80</td>
</tr>
<tr>
<td>DCBREL:</td>
<td>DCBRELAD:00000000 DCBRELA:00000000</td>
</tr>
<tr>
<td>JFCB:</td>
<td>00000000_00009C08</td>
</tr>
<tr>
<td>MBUF:</td>
<td>00000000_00007B20</td>
</tr>
<tr>
<td>FCB:</td>
<td>00000000_00009DF0</td>
</tr>
</tbody>
</table>

Figure 214. Example of formatted C/C++ output from LEDA VERBEXIT (AMODE 64) (Part 8 of 10)
Figure 215. Example of formatted C/C++ output from LEDA VERBEXIT (AMODE 64) (Part 9 of 10)
C/C++-specific sections of the LEDA output

Table 58 describes the contents of the LEDA output that is specific to C/C++.

Table 58. Contents of C/C++-specific sections of the LEDA output (AMODE 64)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] CGEN</td>
<td>Formats the C/C++-specific portion of the Language Environment common anchor area (CAA).</td>
</tr>
<tr>
<td>[2] CGENE</td>
<td>Formats the extension to the C/C++-specific portion of the Language Environment common anchor area (CAA).</td>
</tr>
</tbody>
</table>
Table 58. Contents of C/C++-specific sections of the LEDATA output (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7] File Control Blocks</td>
<td>Formats the C/C++ file control block (FCB). The FCB and its related control blocks, which are listed below, represent the information needed by each open stream.</td>
</tr>
<tr>
<td>FFIL</td>
<td>Formats the header of the C/C++ file control block (FCB).</td>
</tr>
<tr>
<td>FSCE</td>
<td>The file specific category extension control block, which represents the specific type of IO being performed. The following FSCEs may be formatted; other FSCEs will be displayed using a generic overlay.</td>
</tr>
<tr>
<td>HFSF</td>
<td>UNIX file system file</td>
</tr>
<tr>
<td>HSPF</td>
<td>Hiper-Space file</td>
</tr>
<tr>
<td>INTC</td>
<td>Intercept file</td>
</tr>
<tr>
<td>MEMF</td>
<td>Memory file</td>
</tr>
<tr>
<td>OSNS</td>
<td>OS no seek</td>
</tr>
<tr>
<td>OSFS</td>
<td>OS fixed text</td>
</tr>
<tr>
<td>OSVF</td>
<td>OS variable text</td>
</tr>
<tr>
<td>OSUT</td>
<td>OS undefined format text</td>
</tr>
<tr>
<td>TDQF</td>
<td>CICS Transient Data Queue file</td>
</tr>
<tr>
<td>TERM</td>
<td>Terminal file</td>
</tr>
<tr>
<td>VSAM</td>
<td>VSAM file</td>
</tr>
<tr>
<td>OSIO</td>
<td>The OS IO interface control block.</td>
</tr>
<tr>
<td>OSIOE</td>
<td>The OS IO extended interface control block.</td>
</tr>
<tr>
<td>DCB</td>
<td>The data control block; for more information, see z/OS DFSMS Macro Instructions for Data Sets.</td>
</tr>
<tr>
<td>DCBE</td>
<td>The data control block extension; for more information, see z/OS DFSMS Macro Instructions for Data Sets.</td>
</tr>
<tr>
<td>JFCB</td>
<td>The job file control block (JFCB); for more information, see z/OS MVS Data Areas in the z/OS Internet library (<a href="http://www.ibm.com/systems/z/ios/zos/library/bkserv">www.ibm.com/systems/z/ios/zos/library/bkserv</a>).</td>
</tr>
<tr>
<td>JFCBX</td>
<td>The job file control block extension (JFCBX).</td>
</tr>
<tr>
<td>MBUF</td>
<td>The message buffer control block (MBUF).</td>
</tr>
</tbody>
</table>

[8] Memory File Control Blocks

Formats the C/C++ memory file control block (MFCB).

---

**Understanding the PL/I-specific LEDATA output**

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of PL/I-specific control blocks from a system dump when the ALL parameter is specified and PL/I is active in the dump. Figure 217 on page 440 illustrates the PL/I-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option. “PL/I-specific sections of the LEDATA output” on page 445 describes the information contained in the formatted output. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.
Figure 217. Example of formatted PL/I output from LEDA VERBEXIT (AMODE 64) (Part 1 of 10)
Chapter 12. Using Language Environment AMODE 64 debugging facilities

Figure 218. Example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64) (Part 2 of 10)
Figure 219. Example of formatted PL/I output from LEDA VERBEXIT (AMODE 64) (Part 3 of 10)
Figure 220. Example of formatted PL/I output from LEDA VERBEXIT (AMODE 64) (Part 4 of 10)
Figure 221. Example of formatted PL/I output from LEDA VERBEXIT (AMODE 64) (Part 9 of 10)
### Table 59. Contents of PL/I-specific sections of the LEDATA output (AMODE 64)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] PCB</td>
<td>This section formats the Enterprise PL/I process-level control block (PCB).</td>
</tr>
<tr>
<td>[2] DYNLST</td>
<td>This section formats the Enterprise PL/I process-level dynamic allocation parameter list.</td>
</tr>
<tr>
<td>[3] TCA</td>
<td>This section formats the Enterprise PL/I task communication control block (TCA).</td>
</tr>
<tr>
<td>[4] FECB</td>
<td>This section formats the PL/I for MVS and VM fetch control block (FECB).</td>
</tr>
<tr>
<td>[5] OCA</td>
<td>This section formats the Enterprise PL/I ON communications control block (OCA).</td>
</tr>
<tr>
<td>[6] PFO</td>
<td>This section formats the Enterprise PL/I file object control block (PFO).</td>
</tr>
<tr>
<td>[7] SHAD_FCO</td>
<td>This section formats the Enterprise PL/I shadow file object control block (shadow FCO).</td>
</tr>
<tr>
<td>[8] FCO</td>
<td>This section formats the Enterprise PL/I file control block (FCO).</td>
</tr>
<tr>
<td>[9] SCB</td>
<td>This section formats the Enterprise PL/I stream I/O control block (SCB).</td>
</tr>
</tbody>
</table>
Table 59. Contents of PL/I-specific sections of the LEDATA output (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[10] ATTRS</td>
<td>This section formats the Enterprise PL/I file attribute map (ATTRS).</td>
</tr>
<tr>
<td>[12] DCB</td>
<td>This section formats the Enterprise PL/I data control block (DCB).</td>
</tr>
</tbody>
</table>

Understanding the AUTH LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of Preinitialized Environments for Authorized Programs-specific control blocks from a system dump when the AUTH parameter is specified. Figure Figure 223 on page 447 illustrates the output produced when the LEDATA VERBEXIT is invoked with the AUTH parameter. Ellipses are used to summarize some sections of the dump. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.
Figure 223. Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 4)
Figure 224. Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 2 of 4)
Routine Control Blocks

Queue #: 0000000000000017

Routine: ALEM001
  ALRI: 00000001_00200140
  +000000 ID:ALRI  Flags:80000000  InstanceNum:00000000 00000022
  +000010 NEXT:00000001_00200340  ALEC:00000000_7F6F7000
  +000020 ALRIAddr:00000000_00000000  AleiInstanceNum:00000000_00000000
  +000030 DNName:........  RoutineNamePtr:00000001_00200320
  +000040 RoutineNameLen:00000000 00000008  RoutineAddr:00000000_263840C0
  +000050 QSTRTAddr:00000000_26384000  DllKeyPtr:00000000_00000000
  +000060 DIIKeyLen:00000000 00000000  ParmLen:00000000
  +000070 FuncEnv:00000000_00000000  MasterAlri:00000000_00000000
  +000080 Ales:00000001_00107CD0
  +0000F8 ENVIType:00000000  EnvAlris:00000000_00000000
  +000100 NextEnvAlri:00000000_00000000

Routine: ALEM001
  ALRI: 00000001_00200140
  +000000 ID:ALRI  Flags:80000000  InstanceNum:00000000 00000022
  +000010 NEXT:00000001_00200340  ALEC:00000000_7F6F7000
  +000020 ALRIAddr:00000000_00000000  AleiInstanceNum:00000000_00000000
  +000030 DNName:........  RoutineNamePtr:00000001_00200320
  +000040 RoutineNameLen:00000000 00000008  RoutineAddr:00000000_263840C0
  +000050 QSTRTAddr:00000000_26384000  DllKeyPtr:00000000_00000000
  +000060 DIIKeyLen:00000000 00000000  ParmLen:00000000
  +000070 FuncEnv:00000000_00000000  MasterAlri:00000000_00000000
  +000080 Ales:00000001_00107CD0
  +0000F8 ENVIType:00000000  EnvAlris:00000000_00000000
  +000100 NextEnvAlri:00000000_00000000

Figure 225. Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 3 of 4)
Figure 226. Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 4 of 4)
Table 60. Contents of AUTH LEDATA VERBEXIT formatted output (AMODE 64)

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] ALEC</td>
<td>Anchor control block for all other Preinitialized Environments for Authorized Programs control blocks within the address space. The ALEC is located from the ASXB (Address Space Extension Block).</td>
</tr>
<tr>
<td>[2] Load Module Control Blocks</td>
<td>Formatted representation of a table of ALMI control blocks. Each ALMI represents a module that was loaded by Preinitialized Environments for Authorized Programs.</td>
</tr>
<tr>
<td>[3] User Managed Control Blocks</td>
<td>Control blocks for all user-managed environments. A user-managed environment is initialized when the CELAAUTH macro is invoked with REQUEST=USERINIT.</td>
</tr>
<tr>
<td>[4] ALEI</td>
<td>Each ALEI control block represents one environment. This is a control block for one user-managed environment. This section is repeated for each user-managed environment that was initialized.</td>
</tr>
<tr>
<td>[5] Routine Control Blocks</td>
<td>Formatted representation of a table of ALRI control blocks. Each ALRI in this section represents a routine that was called by the user-managed environment. Each ALRI appears in the table twice, once for the routine name and once for the routine address. This is a control block for one user-managed environment. This section is repeated for each user-managed environment that was initialized.</td>
</tr>
<tr>
<td>[6] System Managed Control Blocks</td>
<td>Control blocks for all system-managed environments. A set of system-managed environments is initialized when the CELAAUTH macro is invoked with REQUEST=MNGDINIT.</td>
</tr>
<tr>
<td>[7] ALES</td>
<td>Each ALES represents a set of system-managed environments. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized.</td>
</tr>
<tr>
<td>[8] ETINDEX and ALESETE</td>
<td>The ETINDEX is the environment definition entry index value and the ALESETE represents the environment definition entry. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized. This is a control block for one environment definition entry. This section is repeated for every environment definition entry (AEDE) that was specified when the set of system-managed environments was initialized.</td>
</tr>
<tr>
<td>[9] Routine Control Blocks</td>
<td>Formatted representation of a table of ALRI control blocks. Each ALRI in this section represents a routine that was called in one of the environments associated with the ETINDEX and ALESETE. Each ALRI appears in the table twice, once for the routine name and once for the routine address. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized. This is a control block for one environment definition entry. This section is repeated for every environment definition entry (AEDE) that was specified when the set of system-managed environments was initialized.</td>
</tr>
</tbody>
</table>
### Table 60. Contents of AUTH LEDATA VERBEXIT formatted output (AMODE 64)  (continued)

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[10] ALEI</td>
<td>Each ALEI control block represents one environment. The ALEIs in this section represent system-managed environments. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized. This is a control block for one environment definition entry. This section is repeated for every environment definition entry (AEDE) that was specified when the set of system-managed environments was initialized. This is a control block for one system-managed environment. This section is repeated for every environment that is associated with the ETINDEX and ALESETE.</td>
</tr>
<tr>
<td>[11] ALRI</td>
<td>Contains the ALRI control blocks for each routine that was called in the environment that was identified by the ALEI. This section does not appear if the environment was not used to call a routine. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized. This is a control block for one environment definition entry. This section is repeated for every environment definition entry (AEDE) that was specified when the set of system-managed environments was initialized. This is a control block for one system-managed environment. This section is repeated for every environment that is associated with the ETINDEX and ALESETE.</td>
</tr>
</tbody>
</table>

### Formatting individual control blocks

In addition to the full LEDATA output which contains many formatted control blocks, the IPCS Control block formatter can also format individual Language Environment control blocks. The IPCS CBF command can be invoked from the "IPCS Subcommand Entry” screen, option 6 of the "IPCS PRIMARY OPTION MENU". For more information on using the IPCS CBF command, see the "CBFORMAT subcommand” section in [z/OS MVS IPCS Commands](https://www.ibm.com/support/knowledgecenter/SSECGU_1.13.0/com.ibm.zos.v1r13.fsips.doc/_psa/cecpbf.html).

**Syntax**

```
CBF address STRUCTure (cbname)
```

- **address**
  The address of the control block in the dump. This is determined by browsing the dump or running the LEDATA VERBEXIT.

- **cbname**
  The name of the control block to be formatted. The control blocks that can be individually formatted are listed in [Table 61 on page 453](#). In general, the name of each control block is similar to that used by the LEDATA VERBEXIT and is generally found in the control block’s eyecatcher field. However, all control block names are prefixed with CEE to uniquely define the Language Environment control block names to IPCS.
For example, the following command produces the output shown in Figure 227:

```c
CBF 100007B18 struct(CELCAA)
```

---

### Table 61. Language Environment control blocks that can be individually formatted

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELCIB</td>
<td>Condition Information Block</td>
</tr>
<tr>
<td>CELCIBH</td>
<td>Condition Information Block Header</td>
</tr>
<tr>
<td>CELDLLF</td>
<td>DLL Failure Control Block</td>
</tr>
<tr>
<td>CELDSA</td>
<td>Dynamic Storage Area</td>
</tr>
<tr>
<td>CELDSAATR</td>
<td>XPLINK Transition Area</td>
</tr>
<tr>
<td>CELEDB</td>
<td>Enclave Data Block</td>
</tr>
<tr>
<td>CELENSQ</td>
<td>Enclave Level Storage Management</td>
</tr>
<tr>
<td>CELHNQ31</td>
<td>Heap Anchor Node 31-bit</td>
</tr>
<tr>
<td>CELCOM</td>
<td>CEL Exception Manager Communications Area</td>
</tr>
<tr>
<td>CELHPCQ</td>
<td>Thread Level Heap Control Block</td>
</tr>
<tr>
<td>CELLA</td>
<td>Library Anchor Area</td>
</tr>
<tr>
<td>CELCA</td>
<td>Library Communication Area</td>
</tr>
<tr>
<td>CELPCB</td>
<td>Process Control Block</td>
</tr>
<tr>
<td>CELRCB</td>
<td>Region Control Block</td>
</tr>
<tr>
<td>CELSANC</td>
<td>Storage Management Control Block</td>
</tr>
<tr>
<td>CELSTSB</td>
<td>Storage Report Statistics Block</td>
</tr>
</tbody>
</table>

---

**Figure 227. CAA formatted by the CBFORMAT IPCS command**
Table 62. Preinitialized Environments for Authorized Programs control blocks that can be individually formatted

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELALEC</td>
<td>Anchor Block</td>
</tr>
<tr>
<td>CELALEI</td>
<td>Environment Information Block</td>
</tr>
<tr>
<td>CELALES</td>
<td>System Managed Environment Set Block</td>
</tr>
<tr>
<td>CELALMI</td>
<td>Module Information Block</td>
</tr>
<tr>
<td>CELALRI</td>
<td>Routine Information Block</td>
</tr>
</tbody>
</table>

**Requesting a Language Environment trace for debugging**

Language Environment provides an in-storage, wrapping trace facility that can reconstruct the events leading to the point where a dump is taken. Language Environment produces a trace table in its dump report when the TRACE runtime option is set to ON and:

- A thread ends abnormally because of an unhandled condition of severity 2 or greater and the TERMTHDACT runtime option is set to DUMP, UADUMP, TRACE, or UATRACE.
- An application terminates normally and the TRACE runtime option is set to DUMP (the default).

For more information about recording done by the TERMTHDACT runtime option or the TRACE runtime option, see [z/OS Language Environment Programming Reference](#).

The TRACE runtime option activates Language Environment runtime library tracing and controls the size of the trace buffer, the type of trace events to record, and it determines whether a dump containing only the trace table should be unconditionally taken when the application (enclave) terminates. The trace table contents can be written out either upon demand or at the termination of an enclave.

The contents of the Language Environment dump depend on the values set in the TERMTHDACT runtime option. Table 63 summarizes the dump contents that are generated under abnormal termination.

Table 63. TERMTHDACT runtime option settings and dump contents produced (AMODE 64)

<table>
<thead>
<tr>
<th>TERMTHDACT value</th>
<th>Type of dump generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMTHDACT(QUIET)</td>
<td>Language Environment dump containing the trace table only</td>
</tr>
<tr>
<td>TERMTHDACT(MSG)</td>
<td>Language Environment dump containing the trace table only</td>
</tr>
<tr>
<td>TERMTHDACT(TRACE)</td>
<td>Language Environment dump containing the trace table and the traceback</td>
</tr>
<tr>
<td>TERMTHDACT(DUMP)</td>
<td>Language Environment dump containing thread/enclave/process storage and control blocks; the trace table is included as an enclave control block</td>
</tr>
<tr>
<td>TERMTHDACT(UAONLY)</td>
<td>System dump of the user address space and a Language Environment dump that contains the trace table</td>
</tr>
<tr>
<td>TERMTHDACT(UATRACE)</td>
<td>Language Environment dump that contains traceback information, and a system dump of the user address space</td>
</tr>
</tbody>
</table>
Table 63. TERMTHDACT runtime option settings and dump contents produced (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>TERMTHDACT value</th>
<th>Type of dump generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMTHDACT(UADUMP)</td>
<td>Language Environment dump containing thread/enclave/process storage and control blocks (the trace table is included as an enclave control block), and a user address space dump</td>
</tr>
</tbody>
</table>
| TERMTHDACT(UAIMM)    | System dump of the user address space of the original abend or program interrupt that occurred before the Language Environment condition manager processing the condition. Also contains a Language Environment dump, which contains the trace table.  
          | TRAP(ON,NOSPIE) must be in effect. When TRAP(ON,SPIE) is in effect, UAIMM equals UAONLY results. For software raised conditions or signals, UAIMM is the same as UAONLY. |

Under normal termination, with the TRACE runtime option set to DUMP, Language Environment generates a dump containing the trace table only, independent of the TERMTHDACT setting.

Language Environment quiesces all threads that are currently running except for the thread that issued the call to cdump(). When you call cdump() in a multithread environment, only the current thread is dumped. Enclave- and process-related storage could have changed from the time the dump request was issued.

**Locating the trace dump**

If your application is running under TSO or batch, and a CEEDUMP DD is not specified, Language Environment writes the CEEDUMP to the batch log (SYSOUT=* by default). You can change the SYSOUT class by specifying a CEEDUMP DD, or by setting the environment variable, `_CEE_DMPTARG=SYSOUT(x)`, where `x` is the preferred SYSOUT class.

If your application is running under z/OS UNIX and is either running in a child process, or if it is invoked by one of the exec family of functions, the dump is written to the z/OS UNIX file system. Language Environment writes the CEEDUMP to one of the following directories in the specified order:

1. The directory in environment variable _CEE_DMPTARG, if found
2. The current working directory, if the directory is not the root directory (/), the directory is writable, and the CEEDUMP path name does not exceed 1024 characters
3. The directory found in environment variable TMPDIR (an environment variable that indicates the location of a temporary directory if it is not /tmp)
4. The /tmp directory

The name of this file changes with each dump and uses the following format:

```
/path/CEEDUMP.Date.Time.Pid
```

`path` The path determined from the above algorithm.
Date       The date the dump is taken, appearing in the format YYYYMMDD (such as 20040918 for September 18, 2004).

Time       The time the dump is taken, appearing in the format HHMMSS (such as 175501 for 05:55:01 p.m.).

Pid        The process ID the application is running in when the dump is taken.

Using the Language Environment trace table format in a dump report

The Language Environment trace table is established unconditionally at enclave initialization time if the TRACE runtime option is set to ON. All threads in the enclave share the trace table; there is no thread-specific table, nor can the table be dynamically extended or enlarged.

Understanding the trace table entry (TTE)

Each trace table entry is a fixed-length record consisting of a fixed-format portion (containing such items as the timestamp, thread ID, and member ID) and a member-specific portion. The member-specific portion has a fixed length, of which some (or all) can be unused. For information about how participating products use the trace table entry, see the product-specific documentation. The format of the trace table entry is shown in Figure 228.

<table>
<thead>
<tr>
<th>Time of Day</th>
<th>Thread ID</th>
<th>Member ID and flags</th>
<th>Member entry type</th>
<th>Mbr-specific info up to a maximum of 104 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char (8)</td>
<td>Char (8)</td>
<td>Char (4)</td>
<td>Char (4)</td>
<td>Char (104)</td>
</tr>
</tbody>
</table>

Figure 228. Format of the trace table entry (AMODE 64)

Time       The 64-bit value obtained from a store clock (STCK).

Thread ID   The 8-byte thread ID of the thread that is adding the trace table entry.

Member ID and Flags
Contains 2 fields:

Member ID   The 1-byte member ID of the member making the trace table entry, as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>CEL</td>
</tr>
<tr>
<td>03</td>
<td>C/C++</td>
</tr>
<tr>
<td>08</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>Enterprise PL/I</td>
</tr>
<tr>
<td>12</td>
<td>Sockets</td>
</tr>
</tbody>
</table>

Flags       24 flags reserved for internal use.

Member Entry Type
A number that indicates the type of the member-specific trace information.
that follows the field. To uniquely identify the information contained in a specific TTE, you must consider Member ID as well as Member Entry Type.

**Member-Specific Information**

Based on the member ID and the member entry type, this field contains the specific information for the entry, up to 104 bytes. For C/C++, the entry type of 1 is a record that records an invocation of a base C runtime library function. The entry consists of the name of the invoking function and the name of the invoked function. Entry type 2 is a record that records the return from the base library function. It contains the returned value and the value of errno.

**Member-specific information in the trace table entry**

Global tracing is activated by using the LE=n suboption of the TRACE runtime option. This requests all Language Environment members to generate trace records in the trace table. The settings for the global trace events are:

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No global trace</td>
</tr>
<tr>
<td>1</td>
<td>Trace all runtime library (RTL) function entry and exits</td>
</tr>
<tr>
<td>2</td>
<td>Trace all RTL mutex init/destroy and lock/unlock</td>
</tr>
<tr>
<td>3</td>
<td>Trace all RTL function entry and exits, and all mutex init/destroy and lock/unlock</td>
</tr>
<tr>
<td>8</td>
<td>Trace all RTL storage allocation/deallocation</td>
</tr>
</tbody>
</table>

**When LE=1 is specified:** Table 64 shows the C/C++ records that may be generated. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 487.

Table 64. LE=1 entry records (AMODE 64)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000001</td>
<td>Base C Library function Entry</td>
</tr>
<tr>
<td>03</td>
<td>00000002</td>
<td>Base C Library function Exit</td>
</tr>
<tr>
<td>03</td>
<td>00000003</td>
<td>Posix C Library function Entry</td>
</tr>
<tr>
<td>03</td>
<td>00000004</td>
<td>Posix C Library function Exit</td>
</tr>
<tr>
<td>03</td>
<td>00000005</td>
<td>XPLINK Base or Posix C Library function Entry</td>
</tr>
<tr>
<td>03</td>
<td>00000006</td>
<td>XPLINK Base or Posix C Library function Exit</td>
</tr>
</tbody>
</table>

**When LE=2 is specified:** Table 65 shows the Language Environment records that may be generated.

Table 65. LE=2 entry records (AMODE 64)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00000101</td>
<td>LT</td>
<td>A</td>
<td>Latch Acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000102</td>
<td>LT</td>
<td>R</td>
<td>Latch Release</td>
</tr>
<tr>
<td>01</td>
<td>00000103</td>
<td>LT</td>
<td>W</td>
<td>Latch Wait</td>
</tr>
<tr>
<td>01</td>
<td>00000104</td>
<td>LT</td>
<td>AW</td>
<td>Latch Acquire after Wait</td>
</tr>
<tr>
<td>01</td>
<td>00000106</td>
<td>LT</td>
<td>I</td>
<td>Latch Increment (Recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000107</td>
<td>LT</td>
<td>D</td>
<td>Latch Decrement (Recursive)</td>
</tr>
<tr>
<td>01</td>
<td>000002FC</td>
<td>LE</td>
<td>EUO</td>
<td>Latch unowned (not released)</td>
</tr>
</tbody>
</table>
### Table 65. LE=2 entry records (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>000002FD</td>
<td>LE</td>
<td>EO</td>
<td>Latch already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>00000301</td>
<td>MX</td>
<td>A</td>
<td>Mutex acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000302</td>
<td>MX</td>
<td>R</td>
<td>Mutex release</td>
</tr>
<tr>
<td>01</td>
<td>00000303</td>
<td>MX</td>
<td>W</td>
<td>Mutex wait</td>
</tr>
<tr>
<td>01</td>
<td>00000304</td>
<td>MX</td>
<td>AW</td>
<td>Mutex acquire after wait</td>
</tr>
<tr>
<td>01</td>
<td>00000305</td>
<td>MX</td>
<td>B</td>
<td>Mutex busy (Trylock failed)</td>
</tr>
<tr>
<td>01</td>
<td>00000306</td>
<td>MX</td>
<td>I</td>
<td>Mutex increment (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000307</td>
<td>MX</td>
<td>D</td>
<td>Mutex decrement (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000315</td>
<td>MX</td>
<td>IN</td>
<td>Mutex initialize</td>
</tr>
<tr>
<td>01</td>
<td>00000316</td>
<td>MX</td>
<td>DS</td>
<td>Mutex destroy</td>
</tr>
<tr>
<td>01</td>
<td>0000031D</td>
<td>MX</td>
<td>BI</td>
<td>Shared memory lock init</td>
</tr>
<tr>
<td>01</td>
<td>0000031E</td>
<td>MX</td>
<td>BD</td>
<td>Shared memory lock destroy</td>
</tr>
<tr>
<td>01</td>
<td>0000031F</td>
<td>MX</td>
<td>BO</td>
<td>shared memory lock obtain</td>
</tr>
<tr>
<td>01</td>
<td>00000320</td>
<td>MX</td>
<td>BC</td>
<td>Shared memory lock obtain on condition</td>
</tr>
<tr>
<td>01</td>
<td>00000321</td>
<td>MX</td>
<td>BR</td>
<td>Shared memory lock release</td>
</tr>
<tr>
<td>01</td>
<td>00000324</td>
<td>MX</td>
<td>CIN</td>
<td>Call to SMC_INIT</td>
</tr>
<tr>
<td>01</td>
<td>00000325</td>
<td>MX</td>
<td>CSD</td>
<td>Call to SMC_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>00000326</td>
<td>MX</td>
<td>CSO</td>
<td>Shared resource obtain</td>
</tr>
<tr>
<td>01</td>
<td>00000327</td>
<td>MX</td>
<td>CSR</td>
<td>Shared resource release</td>
</tr>
<tr>
<td>01</td>
<td>00000328</td>
<td>MX</td>
<td>CST</td>
<td>Call to SMC_SetupToWait</td>
</tr>
<tr>
<td>01</td>
<td>00000329</td>
<td>MX</td>
<td>CSP</td>
<td>Call to SMC_POST</td>
</tr>
<tr>
<td>01</td>
<td>000004CC</td>
<td>ME</td>
<td>FFR</td>
<td>Error - Forced release (shared mutex)</td>
</tr>
<tr>
<td>01</td>
<td>000004CD</td>
<td>ME</td>
<td>FFD</td>
<td>Error - Forced decrement (shared mutex)</td>
</tr>
<tr>
<td>01</td>
<td>000004CE</td>
<td>ME</td>
<td>FBD</td>
<td>Error - BPX_SMC(DESTROY) error return</td>
</tr>
<tr>
<td>01</td>
<td>000004CF</td>
<td>ME</td>
<td>FBU</td>
<td>Error - BPX_SMC(fail) returns EBUSY</td>
</tr>
<tr>
<td>01</td>
<td>000004D0</td>
<td>ME</td>
<td>FIV</td>
<td>Error - BPX_SMC(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000004D4</td>
<td>ME</td>
<td>FDU</td>
<td>Error - Destroy failed (uninitialized) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000004D5</td>
<td>ME</td>
<td>FP</td>
<td>Error - Program check (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000004DB</td>
<td>ME</td>
<td>ESC</td>
<td>Error - BPX1SMC error return</td>
</tr>
<tr>
<td>01</td>
<td>000004DE</td>
<td>ME</td>
<td>EDL</td>
<td>Shared memory lock returns deadlock</td>
</tr>
<tr>
<td>01</td>
<td>000004DF</td>
<td>ME</td>
<td>EIV</td>
<td>Shared memory lock returns invalid</td>
</tr>
<tr>
<td>01</td>
<td>000004E0</td>
<td>ME</td>
<td>EPM</td>
<td>Shared memory lock returns eperm</td>
</tr>
<tr>
<td>01</td>
<td>000004E1</td>
<td>ME</td>
<td>EAG</td>
<td>Shared memory lock returns eagain</td>
</tr>
<tr>
<td>01</td>
<td>000004E2</td>
<td>ME</td>
<td>EBU</td>
<td>Shared memory lock returns ebusy</td>
</tr>
<tr>
<td>01</td>
<td>000004E3</td>
<td>ME</td>
<td>ENM</td>
<td>Shared memory lock returns enomem</td>
</tr>
<tr>
<td>01</td>
<td>000004E4</td>
<td>ME</td>
<td>EBR</td>
<td>Shared memory lock release error</td>
</tr>
<tr>
<td>01</td>
<td>000004E5</td>
<td>ME</td>
<td>EBC</td>
<td>Shared memory lock obtain condition error</td>
</tr>
<tr>
<td>01</td>
<td>000004E6</td>
<td>ME</td>
<td>EBO</td>
<td>Shared memory lock obtain error</td>
</tr>
</tbody>
</table>
Table 65. LE=2 entry records (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>000004E7</td>
<td>ME</td>
<td>EBD</td>
<td>Shared memory lock destroy error</td>
</tr>
<tr>
<td>01</td>
<td>000004E8</td>
<td>ME</td>
<td>EBI</td>
<td>Shared memory lock initialize error</td>
</tr>
<tr>
<td>01</td>
<td>000004E9</td>
<td>ME</td>
<td>EFR</td>
<td>Mutex forced release</td>
</tr>
<tr>
<td>01</td>
<td>000004EA</td>
<td>ME</td>
<td>EFD</td>
<td>Mutex forced decrement</td>
</tr>
<tr>
<td>01</td>
<td>000004EB</td>
<td>ME</td>
<td>EDD</td>
<td>Mutex destroy failed (damage)</td>
</tr>
<tr>
<td>01</td>
<td>000004EC</td>
<td>ME</td>
<td>EDB</td>
<td>Mutex destroy failed (busy)</td>
</tr>
<tr>
<td>01</td>
<td>000004ED</td>
<td>ME</td>
<td>EIA</td>
<td>Mutex initialize failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000004EE</td>
<td>ME</td>
<td>EIS</td>
<td>Mutex initialize failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000004EF</td>
<td>ME</td>
<td>EF</td>
<td>Mutex release (forced by quiesce)</td>
</tr>
<tr>
<td>01</td>
<td>000004F0</td>
<td>ME</td>
<td>EP</td>
<td>Mutex program check</td>
</tr>
<tr>
<td>01</td>
<td>000004FA</td>
<td>ME</td>
<td>EDU</td>
<td>Mutex destroy failed (uninitialized)</td>
</tr>
<tr>
<td>01</td>
<td>000004FB</td>
<td>ME</td>
<td>EUI</td>
<td>Mutex uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000004FC</td>
<td>ME</td>
<td>EUO</td>
<td>Mutex unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000004FD</td>
<td>E</td>
<td>EO</td>
<td>Mutex already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000004FE</td>
<td>ME</td>
<td>EIN</td>
<td>Mutex initialization failed (duplicate)</td>
</tr>
<tr>
<td>01</td>
<td>00000508</td>
<td>CV</td>
<td>MR</td>
<td>CV release mutex</td>
</tr>
<tr>
<td>01</td>
<td>00000509</td>
<td>CV</td>
<td>MA</td>
<td>CV reacquire mutex</td>
</tr>
<tr>
<td>01</td>
<td>0000050A</td>
<td>CV</td>
<td>MW</td>
<td>CV mutex wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050B</td>
<td>CV</td>
<td>MAW</td>
<td>CV reacquire mutex after wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050C</td>
<td>CV</td>
<td>CW</td>
<td>CV condition wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050D</td>
<td>CV</td>
<td>CTW</td>
<td>CV condition timeout</td>
</tr>
<tr>
<td>01</td>
<td>0000050E</td>
<td>CV</td>
<td>CWP</td>
<td>CV wait posted</td>
</tr>
<tr>
<td>01</td>
<td>0000050F</td>
<td>CV</td>
<td>CWI</td>
<td>CV wait interrupted</td>
</tr>
<tr>
<td>01</td>
<td>00000510</td>
<td>CV</td>
<td>CTO</td>
<td>CV wait timeout</td>
</tr>
<tr>
<td>01</td>
<td>00000511</td>
<td>CV</td>
<td>CSS</td>
<td>CV condition signal success</td>
</tr>
<tr>
<td>01</td>
<td>00000512</td>
<td>CV</td>
<td>CSM</td>
<td>CV condition signal miss</td>
</tr>
<tr>
<td>01</td>
<td>00000513</td>
<td>CV</td>
<td>CBS</td>
<td>CV condition broadcast success</td>
</tr>
<tr>
<td>01</td>
<td>00000514</td>
<td>CV</td>
<td>CBM</td>
<td>CV condition broadcast miss</td>
</tr>
<tr>
<td>01</td>
<td>00000515</td>
<td>CV</td>
<td>IN</td>
<td>CV initialize</td>
</tr>
<tr>
<td>01</td>
<td>00000516</td>
<td>CV</td>
<td>DS</td>
<td>CV destroy</td>
</tr>
<tr>
<td>01</td>
<td>00000522</td>
<td>CV</td>
<td>CIN</td>
<td>Call to SMC_INIT</td>
</tr>
<tr>
<td>01</td>
<td>00000523</td>
<td>CV</td>
<td>CSD</td>
<td>Call to SMC_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>00000529</td>
<td>CV</td>
<td>CSP</td>
<td>Call to SMC_POST</td>
</tr>
<tr>
<td>01</td>
<td>0000052A</td>
<td>CV</td>
<td>CSB</td>
<td>Call to SMC_POSTALL</td>
</tr>
<tr>
<td>01</td>
<td>0000052B</td>
<td>CV</td>
<td>CSW</td>
<td>Call to SMC_WAIT</td>
</tr>
<tr>
<td>01</td>
<td>0000052C</td>
<td>CV</td>
<td>DBM</td>
<td>Shared condition broadcast - miss</td>
</tr>
<tr>
<td>01</td>
<td>0000052D</td>
<td>CV</td>
<td>DBS</td>
<td>Shared condition broadcast - success</td>
</tr>
<tr>
<td>01</td>
<td>0000052E</td>
<td>CV</td>
<td>DDS</td>
<td>Destroy (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>0000052F</td>
<td>CV</td>
<td>DIN</td>
<td>Initialize (shared mutex/CV)</td>
</tr>
</tbody>
</table>
Table 65. LE=2 entry records (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00000530</td>
<td>CV</td>
<td>DSM</td>
<td>Condition signal - miss (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000531</td>
<td>CV</td>
<td>DSS</td>
<td>Condition signal - success (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000532</td>
<td>CV</td>
<td>DWI</td>
<td>Wait interrupted (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000533</td>
<td>CV</td>
<td>DTO</td>
<td>Wait timeout (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000534</td>
<td>CV</td>
<td>DWP</td>
<td>Wait posted (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006CB</td>
<td>CE</td>
<td>FBT</td>
<td>Error - Invalid system TOD (shared)</td>
</tr>
<tr>
<td>01</td>
<td>000006D1</td>
<td>CE</td>
<td>FRM</td>
<td>Error - Recursive mutex (shared)</td>
</tr>
<tr>
<td>01</td>
<td>000006D2</td>
<td>CE</td>
<td>FUO</td>
<td>Error - Shared mutex unowned</td>
</tr>
<tr>
<td>01</td>
<td>000006D3</td>
<td>CE</td>
<td>FDB</td>
<td>Error - Destroy failed (busy) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D4</td>
<td>CE</td>
<td>FDU</td>
<td>Error - Destroy failed (uninitialized) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D5</td>
<td>CE</td>
<td>FP</td>
<td>Error - Program check (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D6</td>
<td>CE</td>
<td>FUI</td>
<td>Error - Shared mutex or CV uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000006D7</td>
<td>CE</td>
<td>ENV</td>
<td>Error - BPX1SMC(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000006D8</td>
<td>CE</td>
<td>EPE</td>
<td>Error - BPX1SMC(fail) returns EPERM</td>
</tr>
<tr>
<td>01</td>
<td>000006D9</td>
<td>CE</td>
<td>EAN</td>
<td>Error - BPX1SMC(fail) returns EAGAIN</td>
</tr>
<tr>
<td>01</td>
<td>000006DA</td>
<td>CE</td>
<td>EIB</td>
<td>Error - BPX1SMC failed (EBUSY)</td>
</tr>
<tr>
<td>01</td>
<td>000006DB</td>
<td>CE</td>
<td>ESC</td>
<td>Error - BPX1SMC failed</td>
</tr>
<tr>
<td>01</td>
<td>000006EB</td>
<td>CE</td>
<td>EDD</td>
<td>CV destroy failed (damage)</td>
</tr>
<tr>
<td>01</td>
<td>000006EC</td>
<td>CE</td>
<td>EDB</td>
<td>CV destroy failed (busy)</td>
</tr>
<tr>
<td>01</td>
<td>000006ED</td>
<td>CE</td>
<td>EIA</td>
<td>CV initialization failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000006EE</td>
<td>CE</td>
<td>EIS</td>
<td>CV initialization failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000006EF</td>
<td>CE</td>
<td>EF</td>
<td>CV forced by quiesce</td>
</tr>
<tr>
<td>01</td>
<td>000006F0</td>
<td>CE</td>
<td>EP</td>
<td>CV program check</td>
</tr>
<tr>
<td>01</td>
<td>000006F1</td>
<td>CE</td>
<td>EBT</td>
<td>CV invalid system TOD</td>
</tr>
<tr>
<td>01</td>
<td>000006F2</td>
<td>CE</td>
<td>EBN</td>
<td>CV invalid timespec (nanoseconds)</td>
</tr>
<tr>
<td>01</td>
<td>000006F3</td>
<td>CE</td>
<td>EBS</td>
<td>CV invalid timespec (seconds)</td>
</tr>
<tr>
<td>01</td>
<td>000006F4</td>
<td>CE</td>
<td>EPO</td>
<td>CV condition post callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F5</td>
<td>CE</td>
<td>ETW</td>
<td>CV condition timed wait callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F6</td>
<td>CE</td>
<td>EWA</td>
<td>CV condition wait callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F7</td>
<td>CE</td>
<td>ESE</td>
<td>CV condition setup callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F8</td>
<td>CE</td>
<td>ERM</td>
<td>CV recursive mutex</td>
</tr>
<tr>
<td>01</td>
<td>000006F9</td>
<td>CE</td>
<td>EWM</td>
<td>CV wrong mutex</td>
</tr>
<tr>
<td>01</td>
<td>000006FA</td>
<td>CE</td>
<td>EDU</td>
<td>CV destroy failed (uninitialized)</td>
</tr>
<tr>
<td>01</td>
<td>000006FB</td>
<td>CE</td>
<td>EUI</td>
<td>CV mutex or CV uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000006FC</td>
<td>CE</td>
<td>EUO</td>
<td>CV mutex unowned</td>
</tr>
<tr>
<td>01</td>
<td>000006FE</td>
<td>CE</td>
<td>EIN</td>
<td>CV initialization failed (duplicate)</td>
</tr>
<tr>
<td>01</td>
<td>00000702</td>
<td>RW</td>
<td>R</td>
<td>Release</td>
</tr>
<tr>
<td>01</td>
<td>00000704</td>
<td>RW</td>
<td>AW</td>
<td>Acquire after wait</td>
</tr>
</tbody>
</table>
Table 65. LE=2 entry records (AMODE 64)  (continued)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00000706</td>
<td>RW</td>
<td>I</td>
<td>Increment (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000707</td>
<td>RW</td>
<td>D</td>
<td>Decrement (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000715</td>
<td>RW</td>
<td>IN</td>
<td>Initialize</td>
</tr>
<tr>
<td>01</td>
<td>00000716</td>
<td>RW</td>
<td>DS</td>
<td>Destroy</td>
</tr>
<tr>
<td>01</td>
<td>00000717</td>
<td>RW</td>
<td>RA</td>
<td>Read acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000718</td>
<td>RW</td>
<td>WA</td>
<td>Write acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000719</td>
<td>RW</td>
<td>RB</td>
<td>Read busy (tryread failed)</td>
</tr>
<tr>
<td>01</td>
<td>0000071A</td>
<td>RW</td>
<td>WB</td>
<td>Write busy (trywrite failed)</td>
</tr>
<tr>
<td>01</td>
<td>0000071B</td>
<td>RW</td>
<td>RW</td>
<td>Read wait</td>
</tr>
<tr>
<td>01</td>
<td>0000071C</td>
<td>RW</td>
<td>WW</td>
<td>Write wait</td>
</tr>
<tr>
<td>01</td>
<td>0000071D</td>
<td>RW</td>
<td>BI</td>
<td>Call to SLK_INIT</td>
</tr>
<tr>
<td>01</td>
<td>0000071E</td>
<td>RW</td>
<td>BD</td>
<td>Call to SLK_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>0000071F</td>
<td>RW</td>
<td>BO</td>
<td>Call to SLK_OBTAIN</td>
</tr>
<tr>
<td>01</td>
<td>00000720</td>
<td>RW</td>
<td>BC</td>
<td>Call to SLK_OBTAIN_COND</td>
</tr>
<tr>
<td>01</td>
<td>00000721</td>
<td>RW</td>
<td>BR</td>
<td>Call to SLK_RELEASE</td>
</tr>
<tr>
<td>01</td>
<td>000008DC</td>
<td>RE</td>
<td>EOW</td>
<td>Error - Already owned for write (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008DD</td>
<td>RE</td>
<td>EOR</td>
<td>Error - Already owned for read (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008DE</td>
<td>RE</td>
<td>EDL</td>
<td>Error - BPX1SLK(fail) returns EDEADLK</td>
</tr>
<tr>
<td>01</td>
<td>000008DF</td>
<td>RE</td>
<td>EIV</td>
<td>Error - BPX1SLK(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000008E0</td>
<td>RE</td>
<td>EPM</td>
<td>Error - BPX1SLK(fail) returns EPERM</td>
</tr>
<tr>
<td>01</td>
<td>000008E1</td>
<td>RE</td>
<td>EAG</td>
<td>Error - BPX1SLK(fail) returns EAGAIN</td>
</tr>
<tr>
<td>01</td>
<td>000008E2</td>
<td>RE</td>
<td>EBS</td>
<td>Error - BPX1SLK(fail) returns EBUSY</td>
</tr>
<tr>
<td>01</td>
<td>000008E3</td>
<td>RE</td>
<td>ENM</td>
<td>Error - BPX1SLK(fail) returns ENOMEM</td>
</tr>
<tr>
<td>01</td>
<td>000008E4</td>
<td>RE</td>
<td>EBR</td>
<td>Error - BPX1SLK(RELEASE) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E5</td>
<td>RE</td>
<td>EBC</td>
<td>Error - BPX1SLK( OBTAIN_COND) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E6</td>
<td>RE</td>
<td>EBO</td>
<td>Error - BPX1SLK( OBTAIN) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E7</td>
<td>RE</td>
<td>EBD</td>
<td>Error - BPX1SLK(DESTROY) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E8</td>
<td>RE</td>
<td>EBI</td>
<td>Error - BPX1SLK(INIT) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E9</td>
<td>RE</td>
<td>EFR</td>
<td>Error - Forced release</td>
</tr>
<tr>
<td>01</td>
<td>000008EA</td>
<td>RE</td>
<td>EFD</td>
<td>Error - Forced decrement</td>
</tr>
<tr>
<td>01</td>
<td>000008ED</td>
<td>RE</td>
<td>EIA</td>
<td>Error - Initialization failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000008EE</td>
<td>RE</td>
<td>EIS</td>
<td>Error - Initialization failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000008EF</td>
<td>RE</td>
<td>EF</td>
<td>Error - Forced by quiesce</td>
</tr>
<tr>
<td>01</td>
<td>000008F0</td>
<td>RE</td>
<td>EP</td>
<td>Error - Program check</td>
</tr>
<tr>
<td>01</td>
<td>000008FB</td>
<td>RE</td>
<td>EUI</td>
<td>Error - Uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000008FC</td>
<td>RE</td>
<td>EUO</td>
<td>Error - Unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000008FD</td>
<td>RE</td>
<td>EO</td>
<td>Error - Already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008FE</td>
<td>RE</td>
<td>EIN</td>
<td>Error - Initialization failed (duplicate)</td>
</tr>
</tbody>
</table>
Table 66 shows the format for the Mutex – Condition Variable – Latch entries in the trace table.

Table 66. Format of the mutex/CV/latch records (AMODE 64)

<table>
<thead>
<tr>
<th>Record fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
</tr>
<tr>
<td>unused</td>
</tr>
</tbody>
</table>

**Class**
- Two character EBCDIC representation of the trace class.
  - LT  Latch
  - LE  Latch Exception
  - MX  Mutex
  - ME  Mutex Exception
  - CV  Condition Variable
  - CE  Condition Variable Exception

**Source**
- One character EBCDIC representation of the event.
  - C  C/C++

**Blank**
- Blank character

**Event**
- Two character EBCDIC representation of the event; see Table 65 on page 457.

**Object addr**
- Fullword address of the mutex object.

**Name 1**
- Optional eight character field containing the name of the function or object to be recorded.

**Name 2**
- Optional eight character field containing the name of the function or object to be recorded.

**When LE=3 is specified:** The trace table will include the records generated by both LE=1 and LE=2.

**When LE=8 is specified:** As Table 67 shows, the trace table will contain only storage allocation records. Currently, this is only supported by C/C++. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 487.

Table 67. LE=8 entry records (AMODE 64)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000005</td>
<td>Storage allocation entry</td>
</tr>
<tr>
<td>03</td>
<td>00000006</td>
<td>Storage allocation exit</td>
</tr>
</tbody>
</table>

**Sample dump for the trace table entry**

Figure 229 on page 463 shows an example of a dump of the trace table when you specify the LE=1 suboption (the library call/return trace).
Requesting a UNIX System Services syscall trace for debugging

Signal SIGTRACE can be sent to a process or process group to start or stop a trace of the z/OS UNIX System Services syscalls made by the application. The signal is implemented as a toggle. With the trace turned on, the z/OS UNIX System Services kernel gathers the syscall trace records for the targeted processes. A system dump of the user address space can be generated by sending signal SIGDUMP to the same process or process group to capture the trace output. See Z/OS UNIX System Services Command Reference for more information about the SIGTRACE signal.
Chapter 13. Debugging AMODE 64 C/C++ routines

This section provides specific information to help you debug AMODE 64 applications that contain one or more C/C++ routines. It includes the following topics:

- Debugging C/C++ I/O routines
- Using XL C/C++ compiler listings
- Generating a Language Environment dump of a C/C++ routine
- Finding C/C++ information in a Language Environment dump
- Debugging example of C/C++ routines

There are several debugging features that are unique to C/C++ routines. Before examining the C/C++ techniques to find errors, you might want to consider the following areas of potential problems:

- To prevent errors that may result from differences in LP64 default argument types, you should include function prototypes for all C/C++ function calls. For C/C++ runtime library functions, see [z/OS XL C/C++ Runtime Library Reference](#).

  **Note:** malloc() is an example of a RTL function which needs this prototype to work correctly in LP64 applications.

- If you are using the fetch() function, see [z/OS XL C/C++ Programming Guide](#) to ensure that you are creating the fetchable module correctly.

- If you are using DLLs, see [z/OS XL C/C++ Programming Guide](#) to ensure that you are using the DLL correctly.

- Ensure that the entry point of the load module is CELQSTRT.

- If you suspect that you are using uninitialized storage, you may want to use the STORAGE runtime option.

- You should avoid:
  - Incorrect casting
  - Referencing an array element with a subscript outside the declared bounds
  - Copying a string to a target with a shorter length than the source string
  - Declaring but not initializing a pointer variable, or using a pointer to allocated storage that has already been freed

If a routine exception occurred and you need more information than the condition handler provided, run your routine with the following runtime options, TRAP(ON, NOSPIE) and TERMTHDACT(UAIMM). Setting these runtime options generates a system dump of the user address space of the original abend or program interrupt prior to the Language Environment condition manager processing the condition. After the system dump is taken by the operating system the Language Environment condition manager continues processing.

**Debugging C/C++ programs**

You can use C/C++ conventions such as __amrc and perror() when you debug C/C++ programs.
Using the __amrc and __amrc2 structures to debug input/output

__amrc, a structure defined in stdio.h, can help you determine the cause of errors resulting from an I/O operation, because it contains diagnostic information (for example, the return code from a failed VSAM operation). There are two structures:

- __amrc (defined by type __amrc_type)
- __amrc2 (defined by type __amrc2_type; this structure contains secondary information that C can provide)

Because any I/O function calls, such as printf(), can change the value of __amrc or __amrc2, make sure you save the contents into temporary structures of __amrc_type and __amrc2_type respectively, before dumping them.

Figure 230 on page 467 shows the structure as it appears in stdio.h.
Figure 230. __amrc structure (AMODE 64)

Figure 231. __amrc2 structure (AMODE 64)

typedef struct __amrctype {
    union {
        int __error;
        struct {
            unsigned short __syscode,
            __rc;
        } __abend;
        struct {
            unsigned char __fdbk_fill,
            __rc,
            __ftncd,
            __fdbk;
        } __feedback;
        struct {
            unsigned short __svc99_info,
            __svc99_error;
        } __code;
        unsigned int __RBA;
        unsigned int __last_op;
        struct {
            unsigned int __len_fill; /* __len + 4 */
            unsigned int __len;
            char __str[120];
            unsigned int __parmr0;
            unsigned int __parmr1;
            unsigned int __fill2[2];
            char __str2[64];
        } __msg;
        unsigned char __amrc_noseek_to_seek; /* padding to make __amrc 256 bytes */
        char __amrc_pad[23];
    } __amrc_type;
    int __error2;
    char __pad__error2[4];
    FILE *__fileptr;
    int *reserved(6);
} __amrc2

The error or warning value from an I/O operation is in __error, __abend, __feedback, or __alloc. Look at __last_op to determine how to interpret the __code union.
[2] __error
A structure that contains error codes for certain macros or services your application uses. Look at __last_op to determine the error codes. __syscode is the system abend code.

[3] __abend
A structure that contains the abend code when errno is set to indicate a recoverable I/O abend. __rc is the return code. For more information on abend codes, see z/OS MVS System Codes.

[4] __feedback
A structure that is used for VSAM only. The __rc stores the VSAM register 15, __fdbk stores the VSAM error code or reason code, and __RBA stores the RBA after some operations.

[5] __alloc
A structure that contains errors during fopen or freopen calls when defining files to the system using SVC 99.

[6] __RBA
The RBA value returned by VSAM after an ESDS or KSDS record is written out. For an RRDS, it is the calculated value from the record number. In AMODE 64 applications, you can no longer use the address of _amrc._RBA as the first argument to flocate(). Instead, _amrc._RBA must be placed into an unsigned long in order to make it 8 bytes wide, since flocate() is updated to indicate that size of (unsigned long) must be specified as the key length (second argument).

[7] __last_op
A field containing a value that indicates the last I/O operation being performed by C/C++ at the time the error occurred. These values are shown in Table 68 on page 469.

[8] __msg
May contain the system error messages from read or write operations emitted from the DFSMS/MVS SYNA0AF macro instruction. Because the message can start with a hexadecimal address followed by a short integer, it is advisable to start printing at MSG+6 or greater so the message can be printed as a string. Because the message is not null-terminated, a maximum of 114 characters should be printed. This can be accomplished by specifying a printf format specifier as %.114s.

[9] __rplfdbwd
This field contains feedback information related to a VSAM RLS failure. This is the feedback code from the IFGRPL control block.

[10] __XRBA
This is the 8 byte relative byte address returned by VSAM after an ESDS or KSDS record is written out. For an RRDS, it is the calculated value from the record number. It may be used in subsequent calls to flocate().

This field contains the reason for the switch from QSAM (noseek) to BSAM with NOTE and POINT macros requested (seek) by the XL C/C++ Runtime Library. This field is set when system-level I/O macro processing triggers an ABEND condition. The macro name values (defined in stdio.h) for this field are as follows:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>__AM BSAM_NOSWITCH</td>
<td>No switch was made.</td>
</tr>
</tbody>
</table>
Macro | Definition
---|---
__AM_BSAM_UPDATE | The data set is open for update
__AM_BSAM_BSAMWRITE | The data set is already open for write (or update) in the same C process.
__AM_BSAM_FBS_APPEND | The data set is recfm=FBS and open for append
__AM_BSAM_LRECLX | The data set is recfm=LRECLX (used for VBS data sets where records span the largest blocksize allowed on the device)
__AM_BSAM_PARTITIONED_DIRECTORY | The data set is the directory for a regular or extended partitioned data set
__AM_BSAM_PARTITIONED_INDIRECT | The data set is a member of a partitioned data set, and the member name was not specified at allocation

[12] __error2
A secondary error code. For example, an unsuccessful rename or remove operation places its reason code here.

[13] __fileptr
A pointer to the file that caused a SIGIOERR to be raised. Use an fldata() call to get the actual name of the file.

[14] __reserved
Reserved for future use.

**__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 68 lists __last_op values you could receive and where to look for further information.

Table 68. __last_op values and diagnosis information (AMODE 64)

<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_BLDL</td>
<td>Sets __error with return code from OS BLDL 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>
</tbody>
</table>
Table 68. __last_op values and diagnosis information (AMODE 64)  (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__I0_DEVTYPE</td>
<td>Sets __error with return code from I/O DEVTYPE macro.</td>
</tr>
<tr>
<td>__I0_RDJFCB</td>
<td>Sets __error with return code from I/O RDJFCB macro.</td>
</tr>
<tr>
<td>__I0_TRKCALC</td>
<td>Sets __error with return code from I/O TRKCALC macro.</td>
</tr>
<tr>
<td>__I0_OBTAIN</td>
<td>Sets __error with return code from I/O CAMLST OBTAIN.</td>
</tr>
<tr>
<td>__I0_LOCATE</td>
<td>Sets __error with return code from I/O CAMLST LOCATE.</td>
</tr>
<tr>
<td>__I0_CATALOG</td>
<td>Sets __error with return code from I/O CAMLST CAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>__I0_UNCATALOG</td>
<td>Sets __error with return code from I/O CAMLST UNCAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>__I0_RENAME</td>
<td>Sets __error with return code from I/O CAMLST RENAME.</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 corrupt. 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 RRDS.</td>
</tr>
<tr>
<td>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------</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 KSDS.</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 PATH.</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 KSDS PATH.</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>
<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_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_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_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>
</tbody>
</table>
### Table 68. __last_op values and diagnosis information (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__CMS_LINEWRT</td>
<td>Sets __err to result of LINEWRT macro.</td>
</tr>
<tr>
<td>__CMS_QUERY</td>
<td>__err 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>__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>__LFS_OPEN</td>
<td>Sets __err 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 <a href="http://www.ibm.com/support/docview.wss?rs=150&amp;context=swr0000001011188&amp;uid=swg21212589">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
<tr>
<td>__LFS_CLOSE</td>
<td>Sets __err 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 <a href="http://www.ibm.com/support/docview.wss?rs=150&amp;context=swr0000001011188&amp;uid=swg21212589">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
<tr>
<td>__LFS_READ</td>
<td>Sets __err 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 <a href="http://www.ibm.com/support/docview.wss?rs=150&amp;context=swr0000001011188&amp;uid=swg21212589">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
<tr>
<td>__LFS_WRITE</td>
<td>Sets __err 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 <a href="http://www.ibm.com/support/docview.wss?rs=150&amp;context=swr0000001011188&amp;uid=swg21212589">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
<tr>
<td>__LFS_LSEEK</td>
<td>Sets __err 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 <a href="http://www.ibm.com/support/docview.wss?rs=150&amp;context=swr0000001011188&amp;uid=swg21212589">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
<tr>
<td>__LFS_FSTAT</td>
<td>Sets __err 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 <a href="http://www.ibm.com/support/docview.wss?rs=150&amp;context=swr0000001011188&amp;uid=swg21212589">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
</tbody>
</table>

### Using file I/O tracing to debug C/C++ file I/O problems

You can use file I/O tracing to debug C/C++ file I/O problems. For more information, see [Debugging I/O programs](http://www.ibm.com/support/docview.wss?rs=150&context=swr0000001011188&uid=swg21212589) in [z/OS XL C/C++ Programming Guide](http://www.ibm.com/support/docview.wss?rs=150&context=swr0000001011188&uid=swg21212589).
Displaying an error message with the perror() function  
To find a failing routine, check the return code of all function calls. After you have found the failing routine, use the perror() function after the routine to display the error message. perror() displays the string that you pass to it and an error message corresponding to the value of errno. perror() writes to the standard error stream (stderr). By default, the errno2 value will be appended to the end of the perror() string.

If you do not want the errno2 value appended to the perror() string, set the _EDC_ADD_ERRNO2 environment variable to 0.

Figure 232 is an example of a routine using perror().

```
#include <stdio.h>
int main(void){
    FILE *fp;
    fp = fopen("myfile.dat", "w");
    if (fp == NULL)
        perror("fopen error");
}
```

Figure 232. Example of a routine using perror() (AMODE 64)

Using __errno2() to diagnose application problems  
Use the __errno2() function when diagnosing problems in an application program. This function enables z/OS XL C/C++ application programs to access additional diagnostic information, __errno2 (errnojr), associated with errno. The __errno2 may be set by the z/OS XL C/C++ runtime library, z/OS UNIX callable services, or other callable services. The __errno2 is intended for diagnostic display purposes only and is not a programming interface.

Note: Not all functions set __errno2 when errno is set. In the cases where __errno2 is not set, the __errno2() function may return a residual value. You may use the __err2ad() function to clear __errno2 to reduce the possibility of a residual value being returned.

Figure 233 is an example of a routine using __errno2() and Figure 234 on page 474 shows the sample output from that routine.

```
#pragma runopts(posix(on))
#define _EXT
#include <stdio.h>
#include <errno.h>

int main(void) {
    FILE *f;
    f = fopen("testfile.dat", "r")
    if (f==NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    }
    return 0;
}
```

Figure 233. Example of a routine using __errno2() (AMODE 64)
Figure 234. Sample output of routine using __errno2() (AMODE 64)

Figure 235 is an example of a routine using the environment variable
_EDC_ADD_ERRNO2. Figure 236 shows the sample output from that routine. For more information about _EDC_ADD_ERRNO2, see z/OS XL C/C++ Programming
Guide.

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(void) {
   FILE *fp;
   /* do NOT add errno2 to perror message */
   setenv("_EDC_ADD_ERRNO2", "0", 1);
   fp = fopen("testfile.dat", "r");
   if (fp == NULL)
      perror("fopen() failed");
   return 0;
}
```

Figure 235. Example of a routine using _EDC_ADD_ERRNO2 (AMODE 64)

fopen() failed: EDC5129I No such file or directory.

Figure 236. Sample output of a routine using _EDC_ADD_ERRNO2 (AMODE 64)

Figure 237 on page 475 is an example of a routine using __err2ad() in combination with __errno2(). Figure 238 on page 475 shows the sample output from that routine. For more information about __errno2() and __err2ad(), see z/OS XL C/C++ Runtime Library Reference.
Using C/C++ listings

For a detailed description of available listings, see z/OS XL C/C++ User’s Guide.

Finding variables

You can determine the value of a variable in the routine at the point of interrupt by using the compiled code listing as a guide to its address, then finding this address in the Language Environment dump or system dump. The method you use depends on the storage class of variable.

It is possible for the routine to be interrupted before the value of the variable is placed in the location provided for it. This can explain unexpected values in the dump.

Steps for finding automatic variables

Perform the following steps to find automatic variables in the Language Environment dump or system dump:

1. Determine the name of the automatic variable and the function it is defined in.
   As an example, we will find the variable aa in the function main from the program cdvzero shown in Figure 69 on page 216.

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

#define _EXT
#pragma runopts(posix(on))

int main(void) {
    FILE *f;
    setenv("_EDC_ADD_ERRNO2", "0", 1);
    f = fopen("testfile.dat", "r");
    if (f == NULL) {
        perror("fopen() failed");
        printf("_errno2 = %08x\n", __errno2());
    }
    /* reset errno2 to zero */
    *__errno2ad() = 0x0;
    printf("_errno2 = %08x\n", __errno2());
    f = fopen("testfile.dat", "r");
    if (fp == NULL) {
        perror("fopen() failed");
        printf("_errno2 = %08x\n", __errno2());
    }
    return 0;
}
```

Figure 237. Example of a routine using __err2ad() in combination with __errno2() (AMODE 64)

fopen() failed: EDC5129I No such file or directory.
__errno2 = 05620062
__errno2 = 00000000
fopen() failed: EDC5129I No such file or directory.
__errno2 = 05620062

Figure 238. Sample output of routine using __err2ad() in combination with __errno2() (AMODE 64)
2. From the compiler listing, locate the variable in the storage offset listing:

| aa   | 5823-0:10 | Class = automatic, | Location = 2248(r4), | Length = 4 |

The location is specified as decimal offset (base register). So variable aa is located at register 4 + 2248 (X'8C8').

3. From the Traceback (in the Language Environment dump or in the formatted output from the IPCS VERBEXIT LEDATA CEEDUMP subcommand for a system dump) locate the function:

| DSA Entry E Offset Load Mod Program Unit Service Status DSA Addr E Addr PU Addr PU Offset Comp Date Attributes |
|--------|------------|-------------------|---------------------|------------|
| 00000004 main +00000016 CDIVZERO          |                      |                    |          |

If the base register is R4, the register 4 value is always the DSA address for the function.
If the base register is not R4, the register value must be located from saved registers.
If the Status field indicates Exception, use the saved registers from when the condition occurred. In the Language Environment dump, the saved registers can be found in the Condition information associated with the DSA address in the Condition Information for Active Routines section. In the formatted output from the IPCS VERBEXIT LEDATA CM subcommand for a system dump, the saved registers can be found in the CIBH that has the DSA address as the value for the SV1 field.
If the Status field indicates Call, use the saved registers from the DSA address that appears on the line above the function in the Traceback. In the Language Environment dump, the DSAs can be found in the "Control Blocks for Active Routines" section. In the formatted output from the IPCS VERBEXIT LEDATA 'STACK' subcommand for a system dump, the DSAs can be found in the "DSA backchain" section.

Note: Some functions do not save all registers.

4. Add the register value to the offset of the variable to obtain the address of the variable. In the Language Environment dump, the contents of the variable can be read in the DSA Frame section corresponding to the function the variable is contained in. For a system dump, use the IPCS LIST subcommand to display the storage where the variable is located.

The address for variable aa is X'1082FF080' + X'980' = X'1082FFA00'.

Restriction: The parameter value might never be stored, since the first few parameters might be passed in registers and there might be no need to save them.

Steps for finding C/C++ parameters
The C/C++ parameter list is always located in the caller's DSA at offset 2176 (X'880'). Parameters that are passed in registers are not always stored in the parameter list. The compiler option XPLINK(STOREARGS) can be used to ensure that all parameters are stored in the parameter list.
Perform the following steps to find parameters in the Language Environment
dump or system dump:

1. Determine the name of the parameter and the function it is for. As an example,
we will find the parameter `pp` for the function `funcb` from the program
`cdivzero` shown in Figure 53. C routine with a divide-by-zero error.

2. From the compiler listing, locate the parameter in the storage offset listing:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Offset</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pp</code></td>
<td>5828-0:15</td>
<td>8</td>
</tr>
</tbody>
</table>

3. From the Traceback (in the Language Environment dump or in the formatted
   output from the IPCS VERBEXIT LEDATA 'CEEDUMP' subcommand for a
   system dump) locate the function:

   If the base register is R4, the register 4 value is always the DSA address for the
   function.

   If the base register is not R4, the register value must be located from saved
   registers.

   If the Status field indicates Exception, use the saved registers from when the
   condition occurred. In the Language Environment dump, the saved registers
   can be found in the Condition information associated with the DSA address in
   the 'Condition Information for Active Routines' section. In the formatted
   output from the IPCS VERBEXIT LEDATA 'CM' subcommand for a system
dump, the saved registers can be found in the CIBH that has the DSA address
   as the value for the SV1 field.

   If the Status field indicates Call, use the saved registers from the DSA address
   that appears on the line above the function in the Traceback. In the Language
   Environment dump, the DSAs can be found in the 'Control Blocks for Active
   Routines' section. In the formatted output from the IPCS VERBEXIT LEDATA
   'STACK' subcommand for a system dump, the DSAs can be found in the 'DSA
   backchain' section.

   **Note:** Some functions do not save all registers.

4. Add the register value to the offset of the parameter to obtain the address of
   the parameter. In the Language Environment dump, the contents of the
   parameter can be read in the DSA Frame section corresponding to the function
   that passed the parameter. For a system dump, use the IPCS LIST subcommand
to display the storage where the parameter is located.

   The address for parameter `pp` is X'1082FF080' + X'980' = X'1082FFA00'.

**Steps for finding members of aggregates**

You can define aggregates in any of the storage classes or pass them as parameters
to a called function. The first step is to find the start of the aggregate. You can
compute the start of the aggregate as described in previous sections, depending on
the type of aggregate used.
The aggregate map provided for each declaration in a routine can further assist in finding the offset of a specific variable within an aggregate. Structure maps are generated using the AGGREGATE compiler option. Figure 239 shows an example of an aggregate.

typedef struct {
    int asid;
    void *addr;
    asfAmodeType amode;
} asfTargetRef;
asfTargetRef tempTargetRef;

Figure 239. Example code for structure variables (AMODE 64)

Figure 240 shows an example of aggregate map.

<table>
<thead>
<tr>
<th>Offset Bytes(Bits)</th>
<th>Length Bytes(Bits)</th>
<th>Member Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>asid</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td><em><strong>PADDING</strong></em></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>addr</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>amode</td>
</tr>
<tr>
<td>17</td>
<td>7</td>
<td><em><strong>PADDING</strong></em></td>
</tr>
</tbody>
</table>

Figure 240. Example of aggregate map (AMODE 64)

To find the value of variable tempTargetRef.addr:
1. Locate the automatic variable tempTargetRef in the storage offset listing:

| tempTargetRef | 209-0:209 | Class = automatic, Location = 2264(r4), Length = 24 |

The variable tempTargetRef is located at register 4 + 2264 (X'8D8'). For this example, assume that the register 4 value is X'1082FD3E0'. The result is X'1082FDCC8'(X'1082FD3E0' + X'8D8'). This is the address of the value of the automatic variable tempTargetRef in the dump.

2. Find the offset of addr in the Aggregate Map, shown in Figure 240. The offset is 8. Add the offset from the Aggregate Map to the address of the tempTargetRef variable.

The result is X'1082FDCC0' (X'1082FDCC8' + X'8'). This is the address of the value of tempTargetRef.addr in the dump.

Generating a Language Environment dump of a C/C++ routine

You can use the cdump(), csnap(), and ctrace() C/C++ functions to generate a Language Environment dump of C/C++ routines.
cdump()

If your routine is running under z/OS, you can generate useful diagnostic information by using the cdump() function. cdump() produces a main storage dump with the activation stack. When cdump() is invoked from a user routine, the C/C++ library issues an OS IEATDUMP macro to obtain a dump of virtual storage. You can use the Interactive Problem Control System (IPCS) to format and analyze IEATDUMP dumps.

The DD definition for CEESNAP must include the desired data set name and DCB information:

```
LRECL=4160,  BLKSIZE=4160, and RECFM=FBS
```

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.

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

Support for IEATDUMP dumps using the _cdump function is provided only under z/OS. In addition to a IEATDUMP dump, a Language Environment formatted dump is also taken.

csnap()

The csnap() function produces a condensed storage dump. 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 //CEEDUMP DD statement in JCL.

See the z/OS XL C/C++ Runtime Library Reference for more details about the syntax of these functions.

ctrace()

The ctrace() function produces a traceback and includes the offset addresses from which the calls were made.

Sample C routine that calls cdump()

Figure 241 on page 480 shows a sample C routine that uses the cdump function to generate a dump. Figure 247 on page 483 shows the dump output.
```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void hsigfpe(int);
void hsigterm(int);
void atf1(void);

typedef int (*FuncPtr_T)(void);

int st1 = 99;
int st2 = 255;
int xcount = 0;

int main(void) {
    /*
     * 1) Open multiple files
     * 2) Register 2 signals
     * 3) Register 1 atexit function
     * 4) Fetch and execute a module
     */

    FuncPtr_T fetchPtr;
    FILE* fp1;
    FILE* fp2;
    int rc;
    fp1 = fopen("myfile.data", "w");
    if (!fp1) {
        perror("Could not open myfile.data for write");
        exit(101);
    }
    fprintf(fp1, "record 1\n");
    fprintf(fp1, "record 2\n");
    fprintf(fp1, "record 3\n");

    fp2 = fopen("memory.data", "wb,type=memory");
    if (!fp2) {
        perror("Could not open memory.data for write");
        exit(102);
    }
    fprintf(fp2, "some data");
    fprintf(fp2, "some more data");
    fprintf(fp2, "even more data");

    signal(SIGFPE, hsigfpe);
    signal(SIGTERM, hsigterm);

    rc = atexit(atf1);
    if (rc) {
        fprintf(stderr, "Failed on registration of atexit function atf1\n");
        exit(103);
    }
    fetchPtr = (FuncPtr_T)fetch("MODULE1");
    if (!fetchPtr) {
        fprintf(stderr, "Failed to fetch MODULE1\n");
        exit(104);
    }
    fetchPtr();
    return(0);
}
```

Figure 241. Example C routine using cdump() to generate a dump (AMODE 64) (Part 1 of 2)
void hsigfpe(int sig) {
    ++st1;
    return;
}

void hsigterm(int sig) {
    ++st2;
    return;
}

void atf1() {
    ++xcount;
}

Figure 242. Example C routine using cdump() to generate a dump (AMODE 64) (Part 2 of 2)

Figure 243 shows a fetched C module.

#include <ctest.h>
#pragma linkage(func1, fetchable)
int func1(void) {
    __cdump("This is a sample dump");
    return(0);
}

Figure 243. Fetched module for C routine (AMODE 64)

Sample C++ routine that generates a Language Environment dump

Figure 244 shows a sample C++ routine that uses a protection exception to generate a dump.

#include <iostream.h>
#include <ctest.h>
#include "stack.h"

int main() {
    cout << "Program starting:\n";
    cerr << "Error report:\n";
    Stack<int> x;
    x.push(1);
    cout << "Top value on stack : " << x.pop() << '\n';
    cout << "Next value on stack: " << x.pop() << '\n';
    return(0);
}

Figure 244. Example C++ routine with protection exception generating a dump (AMODE 64)

Figure 245 on page 482 shows the template file stack.c

Chapter 13. Debugging AMODE 64 C/C++ routines 481
Figure 246 shows the header file stack.h.

Sample Language Environment dump with C/C++-specific information

This sample dump was produced by compiling the routines shown in Figure 241 on page 480 and Figure 243 on page 481. They were both compiled using options LP64 and GONUM to produce statement numbers in the CEEDUMP. Notice the sequence of calls in the traceback section - CELQINIT is the Language Environment module that invokes the main entry. main calls fetchPtr() at statement number 60, which in turn, through @@FECBMODULE1 fetches the user-defined function func1 shown in Figure 243 on page 481. func1 calls the library routine __cdump() in statement number 5. The complete program unit names for main and func1 are shown in the Fully Qualified Names section along with its load module name.
Figure 247. Example dump from sample C routine (AMODE 64) (Part 1 of 4)
Figure 248. Example dump from sample C routine (AMODE 64) (Part 2 of 4)
Figure 249. Example dump from sample C routine (AMODE 64) (Part 3 of 4)
Figure 250. Example dump from sample C routine (AMODE 64) (Part 4 of 4)
C/C++ contents of the Language Environment trace tables

Language Environment provides C/C++ trace table entry types 5 and 6, which contain character data.

Trace entry 5 occurs when a C library function is called. The format for trace table entry 5 is:

```
NameOfCallingFunction -->(xxx) NameOfCalledFunction<(input_parameters)>
```

or, for called functions calloc, free, malloc, and realloc:

```
NameOfCallingFunction -->(xxx) NameOfCalledFunction<(input_parameters)>
```

In addition, when the call is due to one of these C++ operators:
- new,
- new[],
- delete,
- delete[]

then the C++ operator will appear and the format becomes:

```
NameOfCallingFunction -->(xxx) NameOfCalledFunction<(input_parameters)>
NameOfC++Operator
```

The input_parameters and NameOfC++Operator only appear for the appropriate functions. The angle brackets (<>) indicate that this information does not always appear.

Trace entry 6 occurs when a C library function returns. The format for trace table entry 6 is:

```
<--(xxx)
R1=xxxxxxxxxxxxxxxx R2=xxxxxxxxxxxxxxxx R3=xxxxxxxxxxxxxxxx
ERRNO=xxxxx ERRNO2=xxxxx
```

In the entry types, (xxx) and (xxxx) are numbers associated with the called library function and are used to associate a specific entry record with its corresponding return record.

For entry types 5 and 6, the number will be the same as the number of the function as seen in the C runtime library definition side-deck, SCEELIB dataset member CELQS003, on the IMPORT statement for that function.

Figure 251 on page 488 shows an XPLINK trace that contains examples of the trace entries 5 and 6. For more information about the Language Environment trace table format, see “Understanding the trace table entry (TTE)” on page 456.
Figure 251. Trace table with XPLINK trace table entries 5 and 6 (AMODE 64)
Debugging examples of C/C++ routines

This section contains examples that demonstrate the debugging process for C/C++ routines. Important areas of the output are highlighted. Data unnecessary to the debugging examples has been replaced by ellipses.

**Divide-by-zero error**

[Figure 253 on page 490] illustrates a C program that contains a divide-by-zero error. The code was compiled with RENT so static and external variables need to be calculated from the WSA field. The code was compiled with LP64, GONUM (to produce statement numbers) and XREF, LIST and OFFSET to generate a listing, which is used to calculate addresses of functions and data. The code was processed by the binder with MAP to generate a binder map, which is used to calculate the addresses of static and external variables. The program was created with the option TERMTHDACT(UADUMP) which produced both a Language environment dump and a system dump.

---

Figure 252. Trace table with XPLINK trace table entries 5 and 6 (AMODE 64)
To debug this routine, use the following steps:

1. Locate the Original Condition message in the Condition Information for Active Routines section of the dump. In this example, the message is CEE3209S. The system detected a fixed-point divide exception. This message indicates the error was caused by an attempt to divide by zero. For additional information about CEE3209S, see z/OS Language Environment Runtime Messages.

The traceback section of the dump indicates that the exception occurred at offset X'52' within function funcb. This information is used along with the compiler-generated Pseudo Assembly Listing to determine where the problem occurred.

If the GONUMBER compiler option is specified, statement number information is in the dump. Figure 254 on page 491 shows the generated traceback from the dump.

Figure 253. C routine with a divide-by-zero error (AMODE 64)

/* C Routine with a Divide-by-Zero Error */
#pragma options(noinline)
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int statint = 73;
int fa;
int funcb(int *pp);
int main(void) {
    int aa, bb=1;
    aa = bb;
    aa = funcb(&aa);
    return(aa);
}
int funcb(int *pp) {
    int result;
    fa = *pp;
    result = fa/(statint-73);
    printf("Result = %d\n",result);
    return result;
}
Figure 254. Sections of the dump from example C/C++ routine (AMODE 64) (Part 1 of 2)
2. In the traceback, statement number 12, corresponding to DSA 7, refers to line:
aa = funcb(&aa); in the listing. This is were entry funcb is called. Similarly,
statement number 18, corresponding to DSA 6, points to line: result =
fa/(statint-73); in the listing. This line is where the divide by zero exception
takes place.

3. Locate the instruction with the divide-by-zero error in the Pseudo Assembly
Listing in Figure 256 on page 493

The offset (within funcb) of the exception from the traceback (X'52') reveals the
divide instruction: DR R6,R0 at that location. Instructions at offsets X'32' through
X'58' refer to the result = fa/(statint-73); line of the C/C++ routine.
Figure 256. Pseudo assembly listing (AMODE 64) (Part 1 of 3)
**Figure 257. Pseudo assembly listing (AMODE 64) (Part 2 of 3)**

*** General purpose registers used: 111101110100000000
*** Floating point registers used: 111111111000000000
*** Size of register spill area: 128(max) 0(used)
*** Size of dynamic storage: 0
*** Size of executable code: 144

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define H 1
#define C 2
#define B 3
#define A 4
#define I 5
#define E 6
#define D 7
#define P 8
#define O 9
#define S 10
#define T 11
#define R 12
#define L 13
#define K 14
#define J 15
#define G 16
#define F 17
#define E 18
#define D 19
#define C 20
#define B 21
#define A 22
#define I 23
#define H 24
#define G 25
#define F 26
#define E 27
#define D 28
#define C 29
#define B 30
#define A 31

int main(void)
{
    int aa, bb=1;
    aa = bb;
    return(aa);
}
```

**Constant Area**

```
| Result | %d.. |
```

**PPA1: Entry Point Constants**

```c
AL1(2) = 1
CE = 2
GPR save mask = 3
'PMA2-PPA1' = 4
'f'-213904399' = 5
Parmlength/4 = 6
Prol len/2; alloc r4 change offset/2 = 7
Code length = 8
Interface mapping flags = 9
Offset to Entry Point Marker = 10
```

```
PPA1 End
```
4. Verify the value of the divisor statint. The procedure specified below is to be used for determining the value of static variables only. If the divisor is an automatic variable, there is a different procedure for finding the value of the variable.

Because this routine was compiled with the RENT option, find the WSA address in the Enclave Control Blocks section of the dump. In this example, this address is X'108300050'. Figure 259 shows the WSA address.

Figure 259. C/C++ CAA information in dump (AMODE 64)

5. Routines compiled with the RENT option must also be processed by the binder. The binder produces the Writable Static Map. Find the offset of statint in the Writable Static Map in Figure 260 on page 496. In this example, the offset is X'30'.
6. Add the WSA address of `X'108300050'` to the offset of `statint`. The result is `X'108300080'`. This is the address of the variable `statint`, which is in the writable static area.

7. Use IPCS to display the writeable static area in the system dump. The value at location `X'108300080'` is `X'49'` (that is, `statint` is 73), and hence the fixed-point divide exception.

Figure 260. Writable static map (AMODE 64)

Listing 0000000108300050 LEN(X'00000100')

Figure 261. IPCS storage display of the writeable static area (AMODE 64)

Calling a nonexistent function

Figure 262 on page 497 demonstrates the error of calling a nonexistent function. This routine was compiled with the compiler options LP64, GONUM, LIST, OFFSET, and RENT and was run with the option TERMTHDACT(UADUMP).
To debug this routine, use the following steps:

1. Locate the Original Condition message in the Condition Information for Active Routines section of the dump, shown in Figure 263 on page 498. In this example, the message is CEE3201S The system detected an operation exception (System Completion Code=0C1). This message suggests that the error was caused by an attempt to branch to an unknown address. For additional information about CEE3201S, see z/OS Language Environment Runtime Messages.

The Location section of the dump indicates that the exception occurred at offset X’-209000D0’ within function funca and that there may have been a bad branch from statement 17 offset X’+00000036’ within function funca. The negative offset indicates that the offset cannot be used to locate the instruction that caused the error. Another indication of bad data is the value of X’00000002’ in the instruction address of the PSW shown in the Condition Information section. This address indicates that an instruction in the routine branched outside the bounds of the routine.

In the traceback, the statement number displayed for entry ‘main’ points to line 12 in the source code shown in Figure 262. This line contains the statement “funca(&aa);” in which entry ‘funca’ is called. As message CEE3841I explains, for entry ‘funca’ no statement number could be displayed. In this example, this problem is caused because ‘funca’ has an invalid offset. For further information about this message see z/OS Language Environment Runtime Messages.
Figure 263. Sections of the dump from example C routine (AMODE 64) (Part 1 of 3)
Figure 264. Sections of the dump from example C routine (AMODE 64) (Part 2 of 3)
Figure 265. Sections of the dump from example C routine (AMODE 64) (Part 3 of 3)

2. Find the branch instructions for funca in the listing in Figure 266. Notice the BASR r7,r6 instruction at offset X'000036'. This branch is part of the instruction aa=func_ptr(); in statement 17 in Figure 262 on page 497.

Figure 266. Pseudo assembly listing (AMODE 64) (Part 1 of 2)
Figure 267. Pseudo assembly listing (AMODE 64) (Part 2 of 2)

3. Find the offset of func_ptr in the Writable Static Map, shown in Figure 268.

---------------------
CLASS C_WSA64
LENGTH = 48
ATTRIBUTES = MAP, DEFER, AMODE=64
OFFSET = 0 IN SEGMENT 002
ALIGN = QDWord
---------------------

CLASS
OFFSET NAME TYPE LENGTH SECTION
0 $PRIV000012 PART 10
10 EXISTSA PART 30 EXISTAC
40 func_ptr PART 8

Figure 268. Writable static map (AMODE 64)

4. Add the offset of func_ptr (X’40’) to the address of WSA (X’10830050’) (the WSA address was obtained from the dump report in Figure 266 on page 500). The result (X’10830090’) is the address of the function pointer func_ptr in the writable static storage area. This value is 0, indicating the variable is uninitialized. Figure 269 on page 502 shows the sections of the dump.

Chapter 13. Debugging AMODE 64 C/C++ routines 501
Handling dumps written to the z/OS UNIX file system

When a z/OS UNIX C/C++ application program is running in an address space created as a result of a call to spawn(), vfork(), or one of the exec family of functions, the SYSMDUMP DD allocation information is not inherited. Even though the SYSMDUMP allocation is not inherited, a SYSMDUMP allocation must exist in the parent in order to obtain a HFS storage dump. If the program terminates abnormally while running in this new address space, the kernel causes an unformatted storage dump to be written to an HFS file in the user's working directory. The file is placed in the current working directory or into /tmp if the current working directory is not defined. The file name has the following format:

/directory/coredump.pid

where directory is the current working directory or tmp, and pid is the hexadecimal process ID (PID) for the process that terminated. For details on how to generate the system dump, see “Steps for generating a system dump in a z/OS UNIX shell” on page 387.

To debug the dump, use the Interactive Problem Control System (IPCS). If the dump was written to an HFS file, you must allocate a data set that is large enough and has the correct attributes for receiving a copy of the HFS file. For example, from the ISPF DATA SET UTILITY panel you can specify a volume serial and data set name to allocate. Doing so brings up the DATA SET INFORMATION panel for specifying characteristics of the data set to be allocated.

Figure 269. IPCS storage display of the writeable static area (AMODE 64)

Figure 269 is a sample filled-in panel that shows the characteristics defined for the URCOMPJRUSL.COREDUMP dump data set. Fill in the information for your data set as shown, and estimate the number of cylinders required for the dump file you are going to copy.
Use the TSO/E OGET or OCOPY command with the BINARY keyword to copy the file into the data set. For example, to copy the HFS memory dump file coredump.00060007 into the data set URCOMP.JRUSL.COREDUMP just allocated, a user with the user ID URCOMP enters the following command:

```
OGET '/u/urcomp/coredump.00060007' 'urcomp.jrusl.coredump' BINARY
```

For more information on using the copy commands, see z/OS UNIX System Services User’s Guide.

After you have copied the memory dump file to the data set, you can use IPCS to analyze the dump. See “Formatting and analyzing system dumps” on page 388 for information about formatting Language Environment control blocks.

---

**Multithreading consideration**

Certain control blocks are locked while a dump is in progress. For example, a csnap() of the file control block would prevent another thread from using or dumping the same information. An attempt to do so causes the second thread to wait until the first one completes before it can continue.

**Understanding C/C++ heap information in storage reports**

Storage reports that contain specific C/C++ heap information can be generated in two ways; details on how to request and interpret the reports are provided in the following sections.

- By setting the Language Environment RPTSTG(ON) runtime option for Language Environment created heaps
- By issuing a stand-alone call to the C function __uheapreport() for user-created heaps.

**Language Environment storage report with heap pools statistics**

To request a Language Environment storage report set RPTSTG(ON). If the C/C++ application specified the HEAPPOOLS(ON) or HEAPPOOLS64(ON) runtime option, the storage report displays heap pools statistics. For a sample storage report with heap pools statistics:
report showing heap pools statistics for a multithreaded C/C++ application, see Figure 160 on page 355. The following sections describe the C/C++ specific heap pools information.

**HEAPPOOLS64 storage statistics**

The HEAPPOOLS64 runtime option controls usage of the heap pools storage algorithm at the enclave level. The heap pools algorithm allows for the definition of one to twelve heap pools, each consisting of a number of storage cells of a specified length.

**Note:** The use of an alternative vendor heap manager (VHM) overrides the use of the HEAPPOOLS64 runtime option.

**HEAPPOOLS64 statistics:**

- **Pool** $p$ size: $ssss$ Get requests: $gggg$
  - $p$ number of the pool. When there are multiple pools for a cell size, the pools are numbered using the format $aa.bbb$
  - $aa$ number for the cell size
  - $bbb$ number for the pool within the cell size
  - $ssss$ cell size specified for the pool
  - $gggg$ number of storage requests that were satisfied from this pool
- **Successful Get Heap requests:** $xxxx-yyyy n$
  - $xxxx$ low side of the 8 byte range
  - $yyyy$ high side of the 8 byte range
  - $n$ number of requests in the 8 byte range
- **Requests greater than the largest cell size** — the number of storage requests that are not satisfied by heap pools.

**Note:** Values displayed in the HEAPPOOLS64 statistics report are not serialized when collected; therefore, the values are not necessarily exact.

**HEAPPOOLS64 summary:** The HEAPPOOLS64 summary displays a report of the HEAPPOOLS64 statistics and provides suggested cell sizes.

**Specified Cell Size**

the size of the cell specified in the HEAPPOOLS64 runtime option

**Element Size**

the size of the cell plus any additional storage needed for control information or to maintain alignment

**Cells Per Extent**

the cell pool count specified by the HEAPPOOLS64 runtime option. When there is more than one pool for a cell size, the count is divided by the number of pools.

**Extents Allocated**

the number of times that each pool allocated an extent in order to optimize storage usage. The extents allocated needs to be either one or two. If the number of extents allocated is too high, increase the cell count for the pool.

**Maximum Cells Used**

the maximum number of cells used for each pool.
Cells In Use
the number of cells that were never freed. A large number in this field
could indicate a storage leak.

Suggested Cell Sizes
sizes that are calculated to optimally use storage (assuming that the
application will __malloc/__free with the same frequency). The suggested
cell sizes are given with no cell counts because the usage of each new cell
pool size is not known. If there are less than 12 cell sizes calculated, then
the last pool size is set at 65536.

For more information about stack and heap storage for AMODE64 applications, see
/z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode

HEAPPOOLS storage statistics
The HEAPPOOLS runtime option controls usage of the heap pools storage
algorithm at the enclave level. The heap pools algorithm allows for the definition
of one to twelve heap pools, each consisting of a number of storage cells of a
specified length. HEAPPOOLS runtime option can be used by AMODE 64
applications to manage user heap storage above the 16MB line and below the 2GB
bar.

Note: The use of an alternative Vendor Heap Manager (VHM) overrides the use
of the HEAPPOOLS runtime option.

HEAPPOOLS statistics:
• Pool \( p \) size: ssss Get requests: gggg
  \( p \) number of the pool. When there are multiple pools for a cell size, the
  pools are numbered using the format aa.bbb
  aa number for the cell size
  bbb number for the pool within the cell size
  ssss cell size specified for the pool
  gggg number of storage requests that were satisfied from this pool
• Successful Get Heap requests: xxxx-yyyy n
  xxxx low side of the 8 byte range
  yyyy high side of the 8 byte range
  n number of requests in the 8 byte range
• Requests greater than the largest cell size — the number of storage requests that
  are not satisfied by heap pools.

Note: Values displayed in the HEAPPOOLS statistics report are not serialized
when collected, therefore the values are not necessarily exact.

HEAPPOOLS summary: The HEAPPOOLS summary displays a report of the
HEAPPOOLS statistics and provides suggested percentages for current cell sizes as
well as suggested cell sizes.
• Specified Cell Size — the size of the cell specified in the HEAPPOOLS runtime
  option
• Element Size — the size of the cell plus any additional storage needed for
  control information or to maintain alignment
• Extent Percent — the cell pool percent specified by the HEAPPOOLS runtime
  option
• Cells Per Extent — the number of cells per extent. This number is calculated using the following formula, with a minimum of four cells:

\[
\text{Initial Heap Size} \times (\text{Extent Percent}/100)/\text{(Element Size)}
\]

**Note:** Having a small number of cells per extent is not suggested because the pool can allocate many extents, which causes the HEAPPOOLS algorithm to perform inefficiently.

• Extents Allocated — the number of times that each pool allocated an extent.

To optimize storage usage, the extents allocated need to be either one or two. If the number of extents allocated is too high, increase the percentage for the pool.

• Maximum Cells Used — the maximum number of cells used for each pool.

• Cells In Use — the number of cells that were never freed.

A large number in this field can indicate a storage leak.

• Suggested Percentages for current Cell Sizes — percentages calculated to find the optimal size of the cell pool extent. The calculation is based on the following formula:

\[
(\text{Maximum Cells Used} \times \text{(Element Size)} \times 100) / \text{Initial Heap Size}
\]

With a minimum of 1% and a maximum of 90%

Make sure that your cell pool extents are neither too large nor too small. If your percentages are too large then additional, unreferenced virtual storage will be allocated, thereby causing the program to exhaust the region size. If the percentages are too small then the HEAPPOOLS algorithm will run inefficiently.

• Suggested Cell Sizes — sizes that are calculated to optimally use storage (assuming that the application will _malloc/_free with the same frequency).

**Note:** The suggested cell sizes are given with no percentages because the usage of each new cell pool size is not known. If there are less than 12 cell sizes calculated and the last calculated cell size is smaller than the largest cell size currently in effect, the largest cell size currently in effect is used for the last suggested cell size.


**C function __uheapreport() storage report**

To generate a user-created heap storage report use the C function, __uheapreport(). Use the information in the report to assist with tuning your application’s use of the user-created heap.

User-created HeapPools statistics

- Pool p size: ssss
  - p — the number of the pool
  - ssss — the cell size specified for the pool.
- Successful Get Heap requests: xxxx-yyyy n
  - xxxx — the low side of the range
  - yyyy — the high side of the range
  - n — the number of requests in the range.
- Requests greater than the largest cell size — the number of storage requests that are not satisfied by heap pools.

Note: Values displayed in the HeapPools statistics report are not serialized when collected, therefore the values are not necessarily exact.

HeapPools summary

The HeapPools summary displays a report of the HeapPool statistics and provides suggested percentages for current cell sizes as well as suggested cell sizes. Figure 271 shows a sample storage report generated by __uheapreport().

- Cell Size — the size of the cell specified on the __ucreate() call
- Cells Per Extent — the cell pool count specified on the __ucreate() call
- Extents Allocated — the number of times that each pool allocated an extent in order to optimize storage usage.
- Maximum Cells Used — the maximum number of cells used for each pool.
- Cells In Use — the number of cells that were never freed.
  A large number in this field could indicate a storage leak.
- Suggested Cell Sizes — sizes that are calculated to optimally use storage (assuming that the application will __umalloc/__ufree with the same frequency).
The suggested cell sizes are given with no cell counts because the usage of each new cell pool size is not known. If there are less than 12 cell sizes calculated, then the last pool size is set at 65536.
Part 4. Appendixes
Appendix A. Diagnosing problems with Language Environment

This section 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 (job or console logs, CICS transient (CESE) queues) for any messages from the failing run.

3. Check the message prefix to identify the system or subsystem that issued the message. This can help you determine the cause of the problem. Following are some of the prefixes and their respective origins.
   - **EDC** The prefix for C/C++ messages. The following series of messages are from the C/C++ runtime 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 runtime component of Language Environment.
   - **FOR** The prefix for messages from the Fortran runtime component of Language Environment.
   - **IBM** The prefix for messages from the PL/I runtime component of Language Environment.
   - **CEE** The prefix for messages from the common runtime component of Language Environment.

4. For any messages received, check for recommendations in the “Programmer Response” sections of the messages in this information.

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.
   - For AMODE 64 applications, IBM recommends that you use the IPCS Verbexit IEDATA with the CEEDUMP option to format the traceback. Check the traceback for the source of the problem. For information on how to generate and use a Language Environment or system dump to isolate the cause of the error, see Chapter 3, “Using Language Environment debugging facilities,” on page 39 or Chapter 12, “Using Language Environment AMODE 64 debugging facilities,” on page 369.
   - Alternatively, in a non-XPLINK environment, you can 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 “Locating the name of the failing routine for a non-XPLINK application.”

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, see 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 binder or linkage editor output listing. Note any changes from the previous successful compilation or run. For an explanation of compiler options, see the compiler-programming guide.

11. If you are experiencing a no-response problem, try to force a dump. For example, CANCEL the program with the dump option.

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.

Locating the name of the failing routine for a non-XPLINK application

If a system dump is taken, follow the save area chain to find out the name of the failing routine and whether IBM owns it. Following are the procedures for locating the name of the failing routine, which is the primary entry point name.

1. Find the entry point associated with the current save area. The entry point address (EPA), located in the previous save area at displacement X’10’, decimal 16, points to it.

2. Determine the entry point type, of which there are four:

<table>
<thead>
<tr>
<th>Entry point type is...</th>
<th>If...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language Environment conforming</td>
<td>The entry point plus 4 is X’00C3C5C5’.</td>
</tr>
<tr>
<td>Language Environment conforming OPLINK</td>
<td>The entry point plus 4 is X’01C3C5C5’. OPLINK linkage conventions are used.</td>
</tr>
<tr>
<td>C/C++</td>
<td>The entry point plus 5 is X’CE’.</td>
</tr>
<tr>
<td>Entry point type is...</td>
<td>If...</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
</tr>
<tr>
<td>Nonconforming</td>
<td>The entry point is none of the above. Nonconforming entry points are for routines that follow the linking convention in which the name is at the beginning of the routine. X’47F0Fxxx’ is the instruction to branch around the routine name.</td>
</tr>
</tbody>
</table>

For routines with Language Environment-conforming and C/C++ entry points, Language Environment provides program prolog areas (PPAs). PPA1 contains the entry point name and the address of the PPA2; PPA2 contains pointers to the timestamp, where release level keyword information is found, and to the PPA1 associated with the primary entry point of the routine.

- If the entry point type of the failing routine is Language Environment-conforming, go to step 3.
- If the entry point type is C/C++, go to step 5.
- If the entry point type is nonconforming, go to step 6 on page 514.

3. If the entry point type is Language Environment-conforming, find the entry point name for the Language Environment or COBOL program.
   a. Use an offset of X’C’ from the entry point to locate the address of the PPA1.
   b. In the PPA1, locate the offset to the length of the name. If OPLINK, then multiply the offset by 2 to locate the actual offset to the length of the name.

   **Note:** Enterprise COBOL V5.1 and later releases use OPLINK.

   c. Add this offset to the PPA1 address to find the halfword containing the length of the name, followed by the entry point name.

   The entry point name appears in EBCDIC, with the translated version in the right-hand margin of the system dump.

4. Find the Language Environment or COBOL program name.
   a. Find the address of PPA2 at X’04’ from the start of PPA1. For Enterprise COBOL V5.1 or later releases, find a signed offset at X’04’ from the start of PPA1, then add this offset to the entry point address to obtain the address of PPA2.
   b. Find the address of the compilation unit's primary entry point at X’10’ in the PPA2. For Enterprise COBOL V5.1 and later releases, find a signed offset at X’10’ in the PPA2, then add this offset to the address of PPA2 to obtain the compilation unit's primary entry point.
   c. Find the entry point name associated with the primary entry point as described above. The primary entry point name is the routine name.

See [z/OS Language Environment Vendor Interfaces](#) for details of:
- the non-XPLINK Language Environment-conforming PPA1 and PPA2.
- the XPLINK Language Environment-conforming PPA1, and the XPLINK PPA1 optional area fields.
- the non-XPLINK Language Environment PPA2.
- the Language Environment PPA2: Compile Unit Block for XPLINK.
- the PPA2 timestamp and version information.

5. If the entry point type is C/C++, find the C/C++ routine name.
   a. Use the entry point plus 4 to locate the offset to the entry point name in the PPA1 (see Figure 272 on page 514).
   b. Use this offset to find the length-of-name byte followed by the routine name.
The routine name appears in EBCDIC, with the translated version in the right-hand margin.

6. If the entry point type is nonconforming, find the PL/I routine name.
   a. Find the one byte length immediately preceding the entry point. This is the length of the routine name.
   b. Go back the number of bytes specified in the name length. This is the beginning of the routine name.

7. If the entry point type is nonconforming, find the name of the routine other than PL/I.
   a. Use the entry point plus 4 as the location of the entry point name.
   b. Use the next byte as the length of the name. The name directly follows the length of name byte. The entry point name appears in EBCDIC with the translated version in the right-hand margin.

Figure 273 shows a nonconforming entry point type. Nonconforming entry points that can appear do not necessarily follow this linking convention. The location of data in these save areas can be unpredictable.

Figure 273. Nonconforming entry point type with sample dump

---

<table>
<thead>
<tr>
<th>C Routine Layout Entry and PPA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>04</td>
</tr>
<tr>
<td>08</td>
</tr>
<tr>
<td>0C</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>yy</td>
</tr>
</tbody>
</table>

Figure 272. C PPA1

---

020000 = 47F0F00C 06D3C9E2 E3C9E300 90ED000C E0B |.00..LISTIT.....|
020010 = 18CF41B0 C29850BD 000850DB 000418DB |....Bq&...&.....|
020020 = 4510C052 E3E8D7D3 C9D54040 01020034 |....TYPLIN.....|
020030 = C200001E C505E3C5 094000FE D4C2C509 |B...ENTER NUMBER|
020040 = 4006C64C D9C5E306 094E240 06D940C1 |OF RECORDS OR A|
020050 = D30000CA 00020058 4510C06C E6C1C9E3 |LL........%WAIT|
020060 = D9C44040 010202F0 E4000000 0ACA0002 |RD.....OU.......|

Figure 273. Nonconforming entry point type with sample dump
Searching the IBM Software Support Database

Failures in the Language Environment product can be described through the use of keywords. A keyword is a descriptive word or abbreviation assigned to describe one aspect of a product failure. A set of keywords, called a keyword string, describes the failure in detail. You can use a keyword or keyword string as a search argument against an IBM software support database, such as the Service Information Search (SIS). The database contains keyword and text information describing all current problems reported through APARs and associated PTFs. IBM Support Center personnel have access to the software support database and are responsible for storing and retrieving the information. Using keywords or a keyword string, they will search the database to retrieve records that describe similar known problems.

If you have IBMLink or some other connection to the IBM databases, you can do your own search for previously recorded product failures before calling the IBM Support Center.

If your keyword or keyword string matches an entry in the software support database, the search may yield a more complete description of the problem and possibly identify a correction or circumvention. Such a search may yield several matches to previously reported problems. Review each error description carefully to determine if the problem description in the database matches the failure.

If a match is not found, go to “Preparing documentation for an authorized program analysis report (APAR).”

Preparing documentation for an authorized program analysis report (APAR)

This section provides an overview of how to prepare documentation if a problem arises. For detailed information, see the Software Support Handbook (www.ibm.com/support/customercare/sas/f/handbook/home.html).

Prepare documentation for an APAR only after you have done the following:

- Eliminated user errors as a possible cause of the problem.
- Followed the diagnostic procedures.
- You or your local IBM Support Center has been unsuccessful with the keyword search.

Having met these criteria, follow the instructions below.

1. Report the problem to IBM.

   If you have not already done so, report the problem to IBM by opening a problem management record (PMR).

   If you have IBMLink or some other connection to IBM databases, you can open a PMR yourself. Or, the IBM Software Support Center can open the PMR after consulting with you on the phone. The PMR is used to document your problem and to record the work that the Support Center does on the problem. Be prepared to supply the following information:

   - Customer number
   - PMR number
   - Operating system
   - Operating system release level
• Your current Language Environment maintenance level (PTF list and list of APAR fixes applied)
• Keyword strings you used to search the IBM software support database
• Processor number (model and serial)
• A description of how reproducible the error is. Can it be reproduced each time? Can it be reproduced only sometimes? Have you been unable to reproduce it? Supply source files, test cases, macros, subroutines, and input files required to re-create the problem. Test cases are not required, but can often speed the response time for your problem.

If the IBM Support Center concludes that the problem described in the PMR is a problem with the Language Environment product, they will work with you to open an APAR, so the problem can be fixed.

2. Provide APAR documentation. When you submit an APAR, you will need to supply information that describes the failure. Table 69 describes how to produce documentation required for submission with the APAR.

Table 69. Problem resolution documentation requirements

<table>
<thead>
<tr>
<th>Item</th>
<th>Materials Required</th>
<th>How to Obtain Materials</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Machine-readable source program, including macros, subroutines, input files, and any other data that might help to reproduce the problem.</td>
<td>IBM-supplied system utility program</td>
</tr>
<tr>
<td>2</td>
<td>Compiler listings:</td>
<td>Use appropriate compiler options</td>
</tr>
<tr>
<td></td>
<td>• Source listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Object listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Storage map</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Traceback</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Cross-reference listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• JCL listing and linkage editor listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Assembler-language expansion</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Dumps</td>
<td>See instructions in Chapter 3, “Using Language Environment debugging facilities,” on page 39 (as directed by IBM support personnel).</td>
</tr>
<tr>
<td></td>
<td>• Language Environment dump</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• System dump</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Partition/region size/virtual storage size</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>List of applied PTFs</td>
<td>System programmer</td>
</tr>
<tr>
<td>6</td>
<td>Operating instructions or console log</td>
<td>Application programmer</td>
</tr>
<tr>
<td>7</td>
<td>JCL statements used to invoke and run the routine, including all runtime options, in machine-readable form</td>
<td>Application programmer</td>
</tr>
<tr>
<td>8</td>
<td>System output associated with the MSGFILE runtime option.</td>
<td>Specify MSGFILE(SYSOUT)</td>
</tr>
<tr>
<td>9</td>
<td>Contents of the applicable catalog</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A hardcopy log of the events leading up to the failure.</td>
<td>Print out each display.</td>
</tr>
</tbody>
</table>

3. Submit the APAR documentation.
When submitting material for an APAR to IBM, carefully pack and clearly identify any media containing source programs, job stream data, interactive environment information, data sets, or libraries.
All magnetic media submitted must have the following information attached and visible:

- The APAR number assigned by IBM.
- A list of data sets on the tape (such as source program, JCL, data).
- A description of how the tape was made, including:
  - The exact JCL listing or the list of commands used to produce the machine-readable source. Include the block size, LRECL, and format of each file. If the file was unloaded from a partitioned data set, include the block size, LRECL, and number of directory blocks in the original data set.
  - Labeling information used for the volume and its data sets.
  - The recording mode and density.
  - The name of the utility program that created each data set.
  - The record format and block size used for each data set.

Any printed materials must show the corresponding APAR number. The IBM service personnel will inform you of the mailing address of the service center nearest you.

If an electronic link with IBM Service is available, use this link to send diagnostic information to IBM Service.

After the APAR is opened and the fix is produced, the description of the problem and the fix will be in the software support database in SIS, accessible through ServiceLink.
Appendix B. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the Contact z/OS web page (www.ibm.com/systems/z/os/zos/webqs.html) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- z/OS TSO/E Primer
- z/OS TSO/E User’s Guide
- z/OS ISPF User’s Guide Vol I

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that the screen reader is set to read out...
All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 \* FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* \* FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element
The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element
The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the...
default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

* indicates an optional syntax element that is repeatable
The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:
1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included
The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.
Notices

This information was developed for products and services that are offered in the USA or elsewhere.

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

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

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

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

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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

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

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.
This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for the Knowledge Centers. Use of hyperlinks in other output formats of this information is at your own risk.

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

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Site Counsel
2455 South Road
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 document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

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

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

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

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

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY,
IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user’s name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at [ibm.com/privacy] and IBM’s Online Privacy Statement at [ibm.com/privacy/details] in the section entitled “Cookies, Web Beacons and Other Technologies,” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at [ibm.com/software/info/product-privacy].

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

* For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (www.ibm.com/software/support/systemsz/lifecycle)
• For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This publication documents information NOT intended to be used as a Programming Interface of Language Environment in z/OS.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and Trademark information (www.ibm.com/legal/copytrade.shtml).

Adobe, Acrobat, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

Other company, product, and service names might be trademarks or service marks of others.
Index

Special characters

__abend 174, 466
__alloc 174, 466
__amrc 174, 466
__cabin() function 360, 367, 387
__code 174, 466
__error 174, 466
__feedback 174, 466
__last_op 174, 466
__le_cib_get() function 360
__le_message_get() function 352, 362
__le_message_get_and_and_write() 362
__le_cib_get() function 352, 362
__msg 175, 468
__reset_exception_handler() function 360
__set_exception_handler() function 360, 361
_BPXK_MDUMP 88, 387
_EDC_ADD_ERRNO2 180, 474

Numerics

64-bit applications 351, 365, 369, 465

A

abend codes
>= 4000 34, 366
< 4000 34, 366
4093 365

abends
internal, table of output 345
Language Environment 36, 341, 367
requested by assembler user exit 27
system 37, 368
under CICS 341
user 36, 368

ABPERC runtime option function 9
generating a system dump and 85
modifying condition handling behavior and 24

ABTERMENC runtime option function 9, 27
using 27

accessibility 519
contact IBM 519
features 519

AGGREGATE compiler option 4, 7
AMODE 64 applications
classifying errors 365
debugging C/C++ 465
preparing for debugging 351
using debugging 369

anywhere heap statistics 21
APAR (Authorized Program Analysis Report) 515
documentation 516
application program interfaces
(API) 360, 362
application programs
debugging
handling a storage dump written
to a BFS file 226, 502
handling a storage dump written
to an HFS file 226, 502
argument in dump 66
arguments, registers, and variables for
active routines 64, 385
assembler language
user exit 26, 27
for CICS 345
generating a system dump
with 85, 387
modifying condition handling
behavior and 27
using 26, 27

atexit
information in dump 200
automatic variables
locating in dump 290, 322

B

base locator
for working storage 245
dump 245
below heap
statistics 21
BLOCKS option of CEE3DMP callable
service 41

C

C library function
trace table entries for 487
C return codes to CICS 342

C/C++ (continued)
dump (continued)
structure variables, locating
in 188
system, structures in 188
file
collection block information 201
status and attributes in dump 201
functions
calling dump, example 190, 478
cdump() 46, 190, 371, 478
csnapt() 46, 190, 192, 371, 478, 479
ctrace() 46, 190, 192, 371, 478, 479
fetch() 173, 465
fopen() 175, 468
ferror() 173, 180, 465, 473
printf() 174, 466
produce dump output 46, 371
memory file control block 201
stdio.h 174, 466
timestamp 190
CAA (common anchor area) 68, 385,
390, 393, 406, 438
call chain 66
CALL statement
CDUMP/CPDUMP 263
DUMP/PDUMP 262
SDUMP 263
callable services 23
case 1 condition token 30, 362
case 2 condition token 30, 362
cdump() function 190, 366, 478
CEE prefix 33, 35, 365, 367
CEE3ABD—terminate enclave with an
abend 24, 36, 86
generating a dump and 86
handling user abends and 24, 36
modifying condition handling
behavior and 24
CEE3DMP—generate dump 39, 66
generating a Language Environment
dump with 39
options 40
relationship to PLIDUMP 282, 316
syntax 40
CEE3GRO—returns location offset 24
CEE3SRP—set resume point 24
CEE3XITA assembler user exit 26, 27
CEE3XITA assembler user exit 345
CEE3DMP—language Environment
Dump Service 39
control blocks 111, 404
locating 160
CEE3DUMP, controlling access 158
CEE3HDLR—register user condition
handler 26
CEE3HDLR—register user exception
handler 361
CEEMGET—get a message 30

© Copyright IBM Corp. 1991, 2018
CEEMOUT—dispatch a message
CEEMRCE—move resume cursor to designated label
CEEMRCR—move resume cursor relative to handle cursor
CEEMSG—get, format, and dispatch a message
CEESGL—signal a condition
CEESTAR—signal a condition
CELQSTR 365
CELYSTR 465
character
  data dump 263
CHECK runtime option
  function 9
  modifying condition handling behavior and 24
CHECKOUT compiler option 4
CICS
  abends 341
  application, from an EXEC CICS command 346
  debugging for 339
  debugging information, table of locations 339
  destination control table (DCT) 339
  example traceback in CESE transient data queue 340
  examples of output 339
  nonzero reason code returned, table of output 345
  reason codes 341
  register and program status word contents 341
  return codes
    Language Environment 342
    runtime messages 339
    transaction dump 340
  rollback 345
class test 238
classifying errors table 33, 365
CLLE (COBOL load list entry) 244
COBCOM control block 248
COBOL
  base locator for working storage
    C-specific common anchor area (C-CAA) 199
  compiler options 5
  debugging examples 248, 254
dump
 /applications in 245
  file information in 245
  linkage section in 245
  local variables in 242
  routine information in 242
  run unit storage in 246
  signal information in 199
  stack frames for active routines in 242
  working storage in 245
diagnosis list 237
listings 240
memory file control block 199
program class storage 245
return codes to CICS 342
COBOL (continued)
  routine
    calling Language Environment dump service 240
COBVEC control block 248
command
  syntax diagrams xvii
COMMAREA (Communication Area) 341
compiler options
  C 3
  COBOL 5
  Fortran 6
  LP64 351, 465
  PL/I 7
condition
  information for active routines 64, 385
    in dump 64, 385
  POSIX 30
  unhandled condition handling behavior, modifying 23
  user-written condition handler 23, 26
  condition information block (CIB) 66, 78
  condition manager 30, 362
  CONDITION option of CEE3DMP callable service 42, 66
  condition token 29, 30, 361
case 1 30, 362
  case 2 30, 362
e续
  example of 30, 362
  conditions, nested 31, 363
  contact
    z/OS 519
control block
  for active routines 65, 385
csntrl() function 192, 479
ctrac() function 192, 366, 479
D
  data
    map listing 240
    values 66
    DCB (data control block) 244
    DCT (destination control table) 339
    DEBUG runtime option 9
dumping
  C, examples 215, 222, 489
  COBOL, examples 248, 254
    for CICS 339
  Fortran, examples 268, 273
  PL/I, examples 294, 299, 326, 334
tool 39, 369
DEPTHHCONDLMT runtime option
  function 9, 352
  modifying condition handling behavior and 25
  wait/loop error and 34
diagnosis checklist 511
display
  errnoj_value 37
DISPLAY statement 28, 237
displays
  errnoj_value 37
dump
  an area of storage 263
date in 60, 382
  dynamically allocated storage in 65
storage
  written to an HPS file 226, 502
  symbolic 263
DUMP suboption of TERMTHDACT runtime option 43, 369
DUMP/PDUMP routine 262
format specifications 262
output 262
usage considerations 262
DYNDUMPS, controlling access 158
E
  EDB (enclave control block) 65, 386
  EDB (enclave data block) 65, 386
  EDC prefix 33, 35, 365, 367
edcmtex text command 37
  EIB (exec interface block) 341
  enclave
    identifier in dump 61, 383
    member list 60, 65, 386
    storage 65
termination
    behavior, establishing 27
  entry information 60, 382
  ENTRY option of CEE3DMP callable service 42
  entry point
    name of active routines in dump 383
  ERRCOUNT runtime option
    function 9
    modifying condition handling behavior and 25
    wait/loop error and 34
  errno 199
  errnoj_value 37
  error
    determining source of 511
    message while Language Environment was handling another error 31, 363
  unanticipated 33, 365
  error
  ESD compiler option 7
  examples
    application abends from 346
    C routines 215, 223, 489
    calling a nonexistent subroutine 250, 297, 330
    COBOL routines 248, 254
    Fortran routines 268, 273
    output under CICS 339
    PL/I routines 294, 299, 326, 334
  SUBSCRIPTRANGE error 248, 294, 326
exception handling
  behavior, modifying 360
  user-written exception handler 360
EXEC CICS DUMP statements 341
external data
for COBOL programs in dump 245

F
feedback xxi
fetch
fetch information in dump 200
fetch() function 173, 465
fetchable module 173, 465
file
for COBOL, in dump 245
status key 238
file control block (FCB) 201
FILES option of CEE3DMP callable service 41
FLAG compiler option 4
floating point registers in dump 61
fopen() function 175, 468
FOR prefix 33, 35
Fortran
compiler options 6
debugging examples 268, 273
dump services 261
ers, determining the source of 259
listings 261

G
general purpose registers 61
GMAREA 244
GONUMBER compiler option 4

H
HANDLE ABEND EXEC CICS command 339
header files, C
ctest.h 190, 478
ermo.h 215, 489
stdio.h 174, 466
stdlib.h 215, 489
heap pool 418
trace report 391
heap pools 119
trace report 124, 424
heap storage
created by CEECRHP callable service 23
in LEDATA Output 114, 408
reports 118, 416
storage in dump 65
user 20
HEAP64 runtime option 354
HEAPCHK runtime option function 9
HEAPZONES runtime option 352
HEAPPOOLS 424
storage statistics 504
HEAPPOOLS64 runtime option 354
HEAPZONE runtime option function 9
HEAPZONES runtime option 352
heap storage
HeapPools
HEADER files, C
HANDLE
H
I/O
conventions 173, 465
IBM prefix 33, 35
IGZ prefix 33, 35
INFOMSGFILTER runtime option function 10, 352
INITIALIZE statement 238
instance specific information (ISI) 362
instruction length counter (ILC) in dump 64, 385
Inter-procedural Analysis (IPA) 351
interactive problem control system (IPCS) analyzing a storage dump 226, 502
cbf command 452
VERBEXIT 389, 390, 393, 428, 439
INTERRUPT compiler option function 7
INTERRUPT runtime option 9, 10
interrupt code in dump 64, 385
IOHEAP64 runtime option 354
ITBLK in dump 248
keyboard
navigation 519
PF keys 519
shortcut keys 519
L
LAA (library anchor area) 453
language constructs 237
Language Environment return codes to CICS 342
summary of changes for V2R1 as updated February 2015 xxiv
summary of changes for V2R3 xxiii, xxiv
symbolic feedback code 29, 361
Language Environment dump 369, 371
C information in 199
CEEIDUMP 9, 39, 352
COBOL information in 244
default options 40
example traceback in 65, 386
Fortran information in 266
multiple enclaves and options
BLOCKS 41
CONDITION 42, 66
ENCLAVE 41
ENTRY 42
FILES 41
FNAME 42
NOBLOCKS 41
NOCONDITION 42
Language Environment dump (continued)
options (continued)
NOENTRY 42
NOFILES 41
NOSTORAGE 42
NOTRACEBACK 41
NOVARIABLES 41
PAGESIZE(n) 42
STACKFRAME 42
STORAGE 41
THREAD 41
TRACEBACK 41, 66
VARIABLES 41, 66
output
for C routines 195
for COBOL program 240
for Fortran routines 266
for PL/I routines 282, 316
information for multiple enclaves 47, 371
PL/I information in 285, 319
section descriptions 59, 382
TERMTHDACT suboptions 44, 370
title 60, 382
traceback with condition information
C routine 195, 482
COBOL program 242
Fortran routine 259, 267
Language Environment routine 59, 382
PL/I routine 282, 316
using C functions 46, 371
using CDUMP/PDUMP subroutine 261
using CEE3DMP callable service 39, 59, 382
dump routines 46, 282, 316
using PLIDUMP subroutine 261
using SDUMP subroutine 240
using DUMP/PDUMP subroutine 261
using PLIDUMP subroutine 46, 282, 316
using SDUMP subroutine 261
using DUMP/PDUMP subroutine 261
using PLIDUMP subroutine 46, 282, 316
using SDUMP subroutine 261
using TERMTHDACT runtime option 42, 369
LCA (library communication area) 453
LEDATA
IPCS Verexit
C/C++ Output 127
COBOL Output 146
Parameters 90
Understanding Output 93
IPCS VERBEXIT 389
C/C++ Output 428
Parameters 390
PL/I Output 439
Understanding Output 393
LIBHEAP64 runtime option 354
linkage section
for COBOL programs in dump 245
LIST compiler option 4, 5, 8
listings generated by compiler
COBOL 240
Fortran 261
PL/I 276, 310
LMESSAGE compiler option 8
local variables 64
Index 531
machine state information
in dump 64, 385
MAP compiler option 5, 8
MEMLIMIT storage parameter 365
memory file control block (MFCB) 199, 201
message
classifying errors and runtime, CICS 339
user-created 28
using in your routine 28
module
fetchable 173, 465
module name prefixes, Language Environment 33, 365
MSG suboption of TERMTHDACT 43, 369
MSGFILE runtime option
function 10
runtime messages and 35, 366
MSGQ runtime option 10

N
navigation
keyboard 519
nested condition 31, 363
no response (wait/loop) 34, 366
NOBLOCKS option of CEE3DMP callable service 41
NOCERROR option of CEE3DMP callable service 42
NOENTRY option of CEE3DMP callable service 42
NOFILES option of CEE3DMP callable service 41
NOSTORAGE option of CEE3DMP callable service 42
NOTRACEBACK option of CEE3DMP callable service 41
NOVARIABLES option of CEE3DMP callable service 41

O
OFFSET compiler option 4, 5, 6, 8
optimizing
C 4, 67
COBOL 5
Fortran 264
PL/I 7
options
C compiler 3
COBOL compiler 5
defaults for dump 44, 370
determining runtime in effect 10, 13, 352
Fortran compiler 6
Language Environment runtime 9, 351
PL/I compiler 7, 8

P
page number
in dump 60, 382
PAGESIZE(n) option of CEE3DMP callable service 42
parameter
checking value of 27
PCB (process-level control block) 445
perror() function 180
perror()function 473
PL/I
address of interrupt, finding in dump 287, 321
CAA address, finding in dump 293, 325
common anchor area (CAA) 293, 325
compiler listings
object code listing 280
static storage map 279
variable storage map 280
compiler options
generating listings with 276, 310
list of 7
CSECT 279
debugging examples 294, 299, 326, 334
dump
error type, finding in 287, 320
parameter list, finding contents in 291, 323
PL/I information, finding in 285, 291, 319, 323
PLIDUMP subroutine and 282, 316
statement number, finding in 287, 321
timestamp, finding in 291
variables, finding in 290, 322
ERROR ON-unit 274, 287, 308, 320
errors 273, 276, 307, 310
floating-point register 274, 308
object code listing 280
ON statement control block 280
static storage listing 279
SUBSCRIPTRANGE condition 274, 275, 294, 296, 297, 308, 309, 326, 330
PLIDUMP subroutine 283, 316
PMR (problem management record) 515
pointer
variable 173, 465
PFA 513
preventive service planning (PSP)
bucket 511
printf() function 28, 174, 466
problem management record (PMR) 515
procedure division listings 240
process
control block 65, 386
member list 60, 386
process control block (PCB) 60, 386
PROFILE runtime option
function 10, 352
program
class storage 245
program prolog area 513
program status word (PSW) 64, 385
PSP (preventive service planning)
bucket 511
Q
QUIET suboption of TERMTHDACT runtime option 43, 369
R
reason code
nonzero returned to CICS 345
under CICS 341
registers 0–15
in dump 61
release number
in dump 60, 382
request parameter list (RPL) 201
return code
bad or nonzero 34, 366
RPTOPTS runtime option 11
RPTSTG runtime option 14, 354
run unit
COBOL 246
level control block 246
storage in dump 65, 246
runtime
messages
under CICS 339
runtime options 24, 360
determining those in effect 10, 13, 352
sample options report 11
specifying 27
S
scope
terminator 237
SDUMP routine
description 263
format specifications 264
output 263
usage considerations 264
sending to IBM
reader comments 221
service routines
CDUMP/CIPDUMP 263
DUMP/PDUMP 262
SDUMP 263
SET statement 238
shortcut keys 519
signal information in dump 199
sorted cross-reference listing 240
SOURCE compiler option 5, 6, 8
stack
frame 66
frame format 66, 67
STACK64 runtime option 354
STACKFRAME option of CEE3DMP 42
callable service 42
statement numbers
in dump 60, 383
static
variables in dump 290, 322
writable map 184, 188, 189
status
of routines in dump 383
stderr 28, 366, 369, 370
stdio.h 174, 466
stdout
storage
evaluating use of 14, 354
for active routines 65, 385
leak detecting 65, 119, 386, 418
report 14, 354
statistics 20, 21
STORAGE compiler option 8
storage dump
written to an HFS file 226, 502
STORAGE option of CEE3DMP callble service 41
STORAGE runtime option 10, 352
structure
map 188
variable example code 188
summary of changes
Language Environment
V2R1 as updated February 2015 xxiv
V2R3 xxiii, xxiv
Summary of changes xxiv
symbolic
feedback code 29, 361
symbolic dumps 263
how to call under Fortran 263
symbolic feedback code
using the 31
syntax diagrams
how to read xvii
system abend
with TRAP(OFF) 34, 366
with TRAP(ON) 34, 366
system dump
generating 85, 386
in z/OS UNIX shell 88

T

task global table (TGT) 244
TERMINAL compiler option 5, 8
TERMTHDACT runtime option
function 9, 42, 297, 330, 351, 369
generating a dump and 25, 360
modifying condition handling behavior and 10, 352
suboptions 43, 369
TEST compiler option 4, 5, 6, 7
TEST runtime option 10, 352
text file name prefixes, Language Environment 33, 365
THDCOM in dump 248
THREAD option of CEE3DMP callble service 41
THREADSTACK64 runtime option 354
time
in dump 60, 382
TRACE runtime option
function 10, 352
trace table 161, 456
TERMTHDACT runtime option 43, 369
TRACEBACK option of CEE3DMP callble service 41, 66
transaction
dump 341
rollback 345
rollback effects of assembler user exit on 345
work area 341
TRAP runtime option
function 10, 352
Language Environment condition handling and 25, 85, 386
Language Environment exception handling and 360
U
UADUMP suboption of TERMTHDACT runtime option 25, 43, 360, 370
UAIMM suboption of TERMTHDACT runtime option 25, 44, 360, 370
UAONLY suboption of TERMTHDACT runtime option 25, 43, 360, 370
UATRACE suboption of TERMTHDACT runtime option 25, 43, 360, 370
unhandled conditions 27, 30, 362
establishing enclave termination behavior for 27
USE EXCEPTION/ERROR declaratives 238
USE FOR DEBUGGING declarative 238, 239
user
abend 36, 367, 368
code 34, 366
exit 26, 27
heap statistics 21
stack
statistics 20
user interface
ISPF 519
TSO/E 519
user-specified abends 36, 367
USRHDLR runtime option 9, 26, 351
utility and service subroutines
CDUMP/CPDUMP 263
DUMP/PDUMP 262
SDUMP 263
V
variables
in Language Environment dump 66
structure example code 188
VARIABLES option of CEE3DMP callble service 41, 66
VBREF compiler option 6
verb cross-reference 240

W
working storage
in dump 65, 245, 385
Writable Static Area 183

X
XPLINK
downward-growing stack 67
finding XPLINK information in a dump 209
stack frame format 66, 67
storage statistics 20
trace table entries for 210
XREF compiler option 6, 8
XUFLOW runtime option
function 10
modifying condition handling behavior and 26

Z
z/OS UNIX System Services
C application program and generating a system dump 226, 502