Before using this information and the product it supports, read the information under "Notices" on page 519.

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

This edition replaces GA22-7560-11.

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

### Figures
- vii

### Tables
- xi

### About this document
- xiii
- Using your documentation
- How to read syntax diagrams
- Symbols
- Syntax items
- Syntax examples
- Where to find more information
- Information updates on the web
- The z/OS Basic Skills Information Center

### How to send your comments to IBM
- xix
- If you have a technical problem

### Summary of Changes
- xxi
- Changes made in z/OS Version 1 Release 13
- Changes made in z/OS Version 1 Release 12
- Changes made in z/OS Version 1 Release 11

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

#### Chapter 1. Preparing your routine for debugging
- 1
- Setting compiler options
- XL C and XL C++ compiler options
- COBOL compiler options
- Fortran compiler options
- PL/I compiler options
- Enterprise PL/I for z/OS compiler options
- Using Language Environment run-time options
- Determining run-time options in effect
- Using the CLER CICS transaction to display and set run-time options
- Controlling storage allocation
- Stack storage statistics
- Heap storage statistics
- HEAPPOOLS storage statistics
- Modifying condition handling behavior
- Language Environment callable services
- Language Environment run-time options
- Customizing condition handlers
- Invoking the assembler user exit
- Establishing enclave termination behavior for unhandled conditions
- Using messages in your routine
- C/C++
- COBOL
- Fortran
- PL/I

### Chapter 2. Classifying errors
- 29
- Identifying problems in routines
- Language Environment module names
- Common errors in routines
- Interpreting run-time messages
- Message prefix
- Message number
- Severity code
- Message text
- Understanding abend codes
- User abends
- System abends
- Using pdump to obtain information about errno2 values
- Format
- Description
- Usage notes
- Message returns
- Examples
- Exit Values

### Chapter 3. Using Language Environment debugging facilities
- 35
- Debug tools
- Language Environment dump service, CEE3DMP
- Generating a Language Environment dump with CEE3DMP
- Generating a Language Environment dump with TERMTHDACT
- Generating a Language Environment dump with language-specific functions
- Understanding the Language Environment dump
- Debugging with specific sections of the Language Environment dump
- Multiple enclave dumps
- Generating a system dump
- Steps for generating a system dump in a batch run-time environment
- Steps for generating a system dump in an IMS run-time environment
- Steps for generating a system dump in a CICS run-time environment
- Steps for generating a Language Environment U4039 abend
- Steps for generating a system dump in a z/OS UNIX shell
- Formatting and analyzing system dumps
- Preparing to use the Language Environment support for IPCS
- Understanding Language Environment IPCS VERBEXIT – LEDATA

© Copyright IBM Corp. 1991, 2011
# Figures

1. Options report example produced by run-time option RPTOPTS(ON) ........................................ 10
2. Storage report produced by run-time option RPTSTG(ON) .................................................. 12
3. Storage report produced by RPTSTG(ON) with XPLINK .................................................. 14
4. Language Environment condition token ........................................................................... 27
5. The C program CELSAMP .................................................................................................. 43
6. The C DLL CELDLL ........................................................................................................ 46
7. Example dump using CEE3DMP ..................................................................................... 47
8. Upward-growing (non-XPLINK) stack frame format ...................................................... 64
9. Downward-growing (XPLINK) stack frame format .......................................................... 65
10. Condition information block .......................................................................................... 75
11. Machine state information block .................................................................................. 78
12. Language Environment dump of multiple enclaves ...................................................... 80
13. Example of formatted output from LEDATA VERBEXIT ............................................. 90
14. Example of formatted PTBL output from LEDATA VERBEXIT .................................... 111
15. Example formatted detailed heap segment report from LEDATA VERBEXIT ............. 113
16. Example formatted detailed HEAPPOOLS report from LEDATA VERBEXIT ............. 119
17. Example formatted detailed HEAPPOOLS trace report from LEDATA VERBEXIT .... 123
18. Example formatted C/C++ output from LEDATA VERBEXIT ......................................... 127
19. Example formatted COBOL output from LEDATA VERBEXIT .......................................... 150
20. Example formatted PL/I output from LEDATA VERBEXIT ............................................ 152
21. CAA formatted by the CBFORMAT IPCS command ........................................................ 160
22. Format of the trace table entry ...................................................................................... 163
23. Trace table in dump output (LE=1 suboption) .............................................................. 171
24. __amrc structure .......................................................................................................... 176
25. __amrc2 structure ......................................................................................................... 177
26. Example of a routine using perror() ........................................................................... 182
27. Example of a routine using __errno2() ....................................................................... 183
28. Sample output of a routine using __errno2() ............................................................... 183
29. Example of a routine using __EDC_ADD_ERRNO2 ....................................................... 183
30. Sample output of a routine using __EDC_ADD_ERRNO2 ................................................ 183
31. Example of a routine using __errno2() in combination with __errno2() ............... 184
32. Sample output of routine using __errno2() in combination with __errno2() ............. 184
33. Writable static map produced by prelinker .................................................................. 187
34. Location of RENT static variable in storage ................................................................. 187
35. Writable static map produced by prelinker .................................................................. 188
36. Location of NORENT static variable in storage ............................................................ 189
37. Example code for parameter variable ........................................................................ 189
38. Example code for parameter variable ........................................................................ 190
39. Partial storage offset listing ....................................................................................... 190
40. Example code for structure variable .......................................................................... 191
41. Example of aggregate map .......................................................................................... 191
42. Writable static map produced by prelinker ................................................................ 192
43. Example C routine using cdump() to generate a dump ............................................ 195
44. Fetched module for C routine ..................................................................................... 196
45. Example C++ routine with protection exception generating a dump ....................... 197
46. Template file STACK.C ............................................................................................... 197
47. Header file STACK.H .................................................................................................. 197
48. Example dump from sample C routine ...................................................................... 199
49. Memory file control block ............................................................................................ 207
50. Registers on entry to CEE3DMP .................................................................................. 207
51. Parameters, registers, and variables for active routines .......................................... 208
52. Condition information for active routines .................................................................. 208
53. Sample XPLINK-compiled program (tranmain) which calls a NOXPLINK-compiled program ................................................................. 209
54. Sample NOXPLINK-compiled program (trandll) which calls an XPLINK-compiled program ........................................................................................................... 209
55. Example dump of calling between XPLINK and non-XPLINK programs ................. 210
56. Trace table with C/C++ trace table entry types 1 thru 4 ......................................... 216
57. Trace table with XPLINK trace table entries 5 and 6 .............................................. 218
58. Trace table with trace table entry types 7 and 8 ......................................................... 219
59. Trace table with trace table entry type 20 .................................................................. 219
60. C routine with a divide-by-zero error ........................................................................ 221
61. Sections of the dump from example C/C++ routine (divide-by-zero error) ............ 222
62. Pseudo assembly listing (C/C++ routine divide-by-zero error) .................................. 223
63. C/C++ CAA information in dump (C/C++ routine divide-by-zero error) .............. 224
64. Writable static map (C/C++ routine divide-by-zero error) ........................................ 224
65. Enclave storage section of dump (C/C++ routine divide-by-zero error) ............... 225
66. C/C++ example of calling a nonexistent subroutine ................................................ 225
67. Sections of the dump from example C routine (calling a nonexistent subroutine) .... 226
68. Pseudo assembly listing (calling a nonexistent subroutine) ...................................... 227
69. Writable static map (calling a nonexistent subroutine) ............................................. 227
70. Enclave control blocks and storage sections in dump (calling a nonexistent subroutine) ............................................................... 228
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>145</td>
<td>Example of formatted output from LEDATA VERBEXIT (AMODE 64)</td>
</tr>
<tr>
<td>146</td>
<td>Example of formatted PTBL output from LEDATA VERBEXIT</td>
</tr>
<tr>
<td>147</td>
<td>Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64)</td>
</tr>
<tr>
<td>148</td>
<td>Example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64)</td>
</tr>
<tr>
<td>149</td>
<td>Example formatted detailed HEAPPOLLS trace report from LEDATA VERBEXIT (AMODE 64)</td>
</tr>
<tr>
<td>150</td>
<td>Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64)</td>
</tr>
<tr>
<td>151</td>
<td>Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64)</td>
</tr>
<tr>
<td>152</td>
<td>CAA formatted by the CBFORMAT IPCS command</td>
</tr>
<tr>
<td>153</td>
<td>Format of the trace table entry (AMODE 64)</td>
</tr>
<tr>
<td>154</td>
<td>Trace table in dump output (LE=1 suboption)</td>
</tr>
<tr>
<td>155</td>
<td>__amrc structure (AMODE 64)</td>
</tr>
<tr>
<td>156</td>
<td>__amrc2 structure (AMODE 64)</td>
</tr>
<tr>
<td>157</td>
<td>Example of a routine using perror() (AMODE 64)</td>
</tr>
<tr>
<td>158</td>
<td>Example of a routine using __errno2() (AMODE 64)</td>
</tr>
<tr>
<td>159</td>
<td>Sample output of routine using __errno2() (AMODE 64)</td>
</tr>
<tr>
<td>160</td>
<td>Example of a routine using _EDC_ADD_ERRNO2 (AMODE 64)</td>
</tr>
<tr>
<td>161</td>
<td>Sample output of a routine using _EDC_ADD_ERRNO2 (AMODE 64)</td>
</tr>
<tr>
<td>162</td>
<td>Example of a routine using __err2ad() in combination with __errno2() (AMODE 64)</td>
</tr>
<tr>
<td>163</td>
<td>Sample output of routine using __err2ad() in combination with __errno2() (AMODE 64)</td>
</tr>
<tr>
<td>164</td>
<td>Example code for structure variables (AMODE 64)</td>
</tr>
<tr>
<td>165</td>
<td>Example of aggregate map (AMODE 64)</td>
</tr>
<tr>
<td>166</td>
<td>Example C routine using cdump() to generate a dump (AMODE 64)</td>
</tr>
<tr>
<td>167</td>
<td>Fetched module for C routine (AMODE 64)</td>
</tr>
<tr>
<td>168</td>
<td>Example C++ routine with protection exception generating a dump (AMODE 64)</td>
</tr>
<tr>
<td>169</td>
<td>Template file STACK.C (AMODE 64)</td>
</tr>
<tr>
<td>170</td>
<td>Header file STACK.H (AMODE 64)</td>
</tr>
<tr>
<td>171</td>
<td>Example dump from sample C routine (AMODE 64)</td>
</tr>
<tr>
<td>172</td>
<td>Trace table with XPLINK trace table entries 5 and 6 (AMODE 64)</td>
</tr>
<tr>
<td>173</td>
<td>C routine with a divide-by-zero error (AMODE 64)</td>
</tr>
<tr>
<td>174</td>
<td>Sections of the dump from example C/C++ routine (AMODE 64)</td>
</tr>
<tr>
<td>175</td>
<td>Pseudo assembly listing (AMODE 64)</td>
</tr>
<tr>
<td>176</td>
<td>C/C++ CAA information in dump (AMODE 64)</td>
</tr>
<tr>
<td>177</td>
<td>Writable static map (AMODE 64)</td>
</tr>
<tr>
<td>178</td>
<td>IPCS storage display of the writable static area (AMODE 64)</td>
</tr>
<tr>
<td>179</td>
<td>C/C++ example of calling a nonexistent subroutine (AMODE 64)</td>
</tr>
<tr>
<td>180</td>
<td>Sections of the dump from example C routine (AMODE 64)</td>
</tr>
<tr>
<td>181</td>
<td>Pseudo assembly listing (AMODE 64)</td>
</tr>
<tr>
<td>182</td>
<td>Writable static map (AMODE 64)</td>
</tr>
<tr>
<td>183</td>
<td>IPCS storage display of the writable static area (AMODE 64)</td>
</tr>
<tr>
<td>184</td>
<td>IPCS panel for entering data set information (AMODE 64)</td>
</tr>
<tr>
<td>185</td>
<td>Storage report generated by __uheapreport() (AMODE 64)</td>
</tr>
<tr>
<td>186</td>
<td>C PPA1</td>
</tr>
<tr>
<td>187</td>
<td>Nonconforming entry point type with sample dump</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>1.</td>
<td>How to Use z/OS Language Environment Publications</td>
</tr>
<tr>
<td>2.</td>
<td>Syntax examples</td>
</tr>
<tr>
<td>3.</td>
<td>TEST suboptions to simplify debugging</td>
</tr>
<tr>
<td>4.</td>
<td>C/C++ compiler options to simplify run-time debugging</td>
</tr>
<tr>
<td>5.</td>
<td>COBOL compiler options for run-time debugging</td>
</tr>
<tr>
<td>6.</td>
<td>Fortran compiler options for run-time debugging</td>
</tr>
<tr>
<td>7.</td>
<td>PL/I compiler options for debugging</td>
</tr>
<tr>
<td>8.</td>
<td>Enterprise PL/I for z/OS compiler options for debugging</td>
</tr>
<tr>
<td>9.</td>
<td>Language Environment run-time options for debugging</td>
</tr>
<tr>
<td>10.</td>
<td>Storage report fields that display stack storage statistics</td>
</tr>
<tr>
<td>11.</td>
<td>Storage report fields that display heap storage statistics</td>
</tr>
<tr>
<td>12.</td>
<td>Callable services that modify condition handling</td>
</tr>
<tr>
<td>13.</td>
<td>Run-time options that modify condition handling</td>
</tr>
<tr>
<td>14.</td>
<td>Callable services that can convert condition tokens to routine variables, messages, or signaled conditions</td>
</tr>
<tr>
<td>15.</td>
<td>Common error symptoms, possible causes, and programmer responses</td>
</tr>
<tr>
<td>16.</td>
<td>CEE3DMP options</td>
</tr>
<tr>
<td>17.</td>
<td>TERMTHDACT suboptions, level of information, and destinations</td>
</tr>
<tr>
<td>18.</td>
<td>Condition handling of 0Cx abends</td>
</tr>
<tr>
<td>19.</td>
<td>Contents of the Language Environment dump</td>
</tr>
<tr>
<td>20.</td>
<td>Description of CAA fields</td>
</tr>
<tr>
<td>21.</td>
<td>Contents of the Language Environment LEDATA VERBEXIT formatted output</td>
</tr>
<tr>
<td>22.</td>
<td>Contents of the Heap report sections of LEDATA output</td>
</tr>
<tr>
<td>23.</td>
<td>Contents of HEAPPOOLS report sections of LEDATA output</td>
</tr>
<tr>
<td>24.</td>
<td>Contents of HEAPPOOLS trace section of LEDATA output</td>
</tr>
<tr>
<td>25.</td>
<td>Contents of C/C++-specific sections of LEDATA output</td>
</tr>
<tr>
<td>26.</td>
<td>Contents of COBOL-specific sections of LEDATA Output</td>
</tr>
<tr>
<td>27.</td>
<td>Contents of PL/I-specific sections of LEDATA output</td>
</tr>
<tr>
<td>28.</td>
<td>Language Environment Control blocks that can be individually formatted</td>
</tr>
<tr>
<td>29.</td>
<td>TERMTHDACT run-time option settings and dump contents produced</td>
</tr>
<tr>
<td>30.</td>
<td>LE=1 entry records</td>
</tr>
<tr>
<td>31.</td>
<td>LE=2 entry records</td>
</tr>
<tr>
<td>32.</td>
<td>Format of the mutex/CV/latch records</td>
</tr>
<tr>
<td>33.</td>
<td>LE=8 entry records</td>
</tr>
<tr>
<td>34.</td>
<td>LE=20 entry records</td>
</tr>
<tr>
<td>35.</td>
<td>__last_op values and diagnosis information</td>
</tr>
<tr>
<td>36.</td>
<td>Finding the WSA base address</td>
</tr>
<tr>
<td>37.</td>
<td>Contents of the C/C++ sections of the Language Environment</td>
</tr>
<tr>
<td>38.</td>
<td>Contents of XPLINK information in a Language Environment dump</td>
</tr>
<tr>
<td>39.</td>
<td>Compiler-generated COBOL listings and their contents</td>
</tr>
<tr>
<td>40.</td>
<td>Compiler-generated Fortran listings and their contents</td>
</tr>
<tr>
<td>41.</td>
<td>Understanding the Language Environment traceback table</td>
</tr>
<tr>
<td>42.</td>
<td>Compiler-generated PL/I for MVS &amp; VM listings and their contents</td>
</tr>
<tr>
<td>43.</td>
<td>Typical comments in a PL/I for MVS &amp; VM static storage listing</td>
</tr>
<tr>
<td>44.</td>
<td>Comments in a PL/I for MVS &amp; VM object code listing</td>
</tr>
<tr>
<td>45.</td>
<td>PL/I for MVS &amp; VM mnemonics</td>
</tr>
<tr>
<td>46.</td>
<td>Compiler-generated PL/I listings and their contents</td>
</tr>
<tr>
<td>47.</td>
<td>Finding data when Language Environment returns a nonzero return code</td>
</tr>
<tr>
<td>48.</td>
<td>Finding data when Language Environment abends internally</td>
</tr>
<tr>
<td>49.</td>
<td>Finding data when Language Environment abends from an EXEC CICS command</td>
</tr>
<tr>
<td>50.</td>
<td>Common error symptoms, possible causes, and programmer responses</td>
</tr>
<tr>
<td>51.</td>
<td>TERMTHDACT suboptions, level of information, and destinations</td>
</tr>
<tr>
<td>52.</td>
<td>Contents of the Language Environment dump - AMODE 64</td>
</tr>
<tr>
<td>53.</td>
<td>Contents of the Language Environment LEDATA VERBEXIT formatted output (AMODE 64)</td>
</tr>
<tr>
<td>54.</td>
<td>Contents of Heap report sections of the LEDATA output (AMODE 64)</td>
</tr>
<tr>
<td>55.</td>
<td>Contents of the Heappool report sections of the LEDATA output (AMODE 64)</td>
</tr>
<tr>
<td>56.</td>
<td>Contents of a detailed HEAPPOOLS trace report from LEDATA VERBEXIT (AMODE 64)</td>
</tr>
<tr>
<td>57.</td>
<td>Contents of C/C++-specific sections of the LEDATA output (AMODE 64)</td>
</tr>
<tr>
<td>58.</td>
<td>Contents of AUTH LEDATA VERBEXIT formatted output (AMODE 64)</td>
</tr>
<tr>
<td>59.</td>
<td>Language Environment control blocks that can be individually formatted</td>
</tr>
<tr>
<td>60.</td>
<td>Preinitialized Environments for Authorized Programs control blocks that can be individually formatted</td>
</tr>
<tr>
<td>61.</td>
<td>TERMTHDACT run-time option settings and dump contents produced (AMODE 64)</td>
</tr>
<tr>
<td>62.</td>
<td>LE=1 entry records (AMODE 64)</td>
</tr>
<tr>
<td>63.</td>
<td>LE=2 entry records (AMODE 64)</td>
</tr>
</tbody>
</table>
64. Format of the mutex/CV/latch records (AMODE 64) ........................................ 461
65. LE=8 entry records (AMODE 64) ................................................................. 461
66. __last_op values and diagnosis information (AMODE 64) .......................... 467
67. Problem resolution documentation requirements ........................................ 512
**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 provided above.

This book is intended for application programmers who are interested in techniques for debugging run-time programs. To use this book, you should be familiar with:

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

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

IBM® z/OS Language Environment (also called Language Environment) provides common services and language-specific routines in a single run-time environment for C, C++, COBOL, Fortran (z/OS only; no support for z/OS UNIX System Services 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 run-time environment for applications 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.

© Copyright IBM Corp. 1991, 2011
Debug Tool is also available as a standalone product. Debug Tool Utilities and Advanced Functions is also available. For more information, see [http://www.ibm.com/software/awdtools/debugtool/](http://www.ibm.com/software/awdtools/debugtool/).

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 run-time libraries for C/C++, COBOL, Fortran, and PL/I.

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

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

Each publication helps you perform different tasks, some of which are listed in Table 1. All books are available in printable (PDF) and BookManager softcopy formats. For a complete list of publications that you may need, see the bibliography on page 523.

**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</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 run-time 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 run-time messages</td>
<td>z/OS Language Environment Run-Time 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 Run-Time 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 Information Center using a screen reader, syntax diagrams are provided in dotted decimal format.

Symbols

The following symbols may be displayed in syntax diagrams:

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

Syntax items

Syntax diagrams contain many different items. Syntax items include:

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

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

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

**Syntax examples**

The following table provides syntax examples.

<table>
<thead>
<tr>
<th>Table 2. Syntax examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Required item.</td>
</tr>
<tr>
<td>Required choice.</td>
</tr>
<tr>
<td>Optional item.</td>
</tr>
<tr>
<td>Optional choice.</td>
</tr>
<tr>
<td>Default.</td>
</tr>
<tr>
<td>Variable.</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></td>
</tr>
<tr>
<td>An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.</td>
<td>![Repeatable Item Diagram]</td>
</tr>
<tr>
<td>A character within the arrow means you must separate repeated items with that character.</td>
<td>![Character Within Arrow Diagram]</td>
</tr>
<tr>
<td>An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected or a single item can be repeated.</td>
<td>![Group Repeatable Items Diagram]</td>
</tr>
</tbody>
</table>

**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 Diagram]

---

### Where to find more information

For an overview of the information associated with z/OS, see [z/OS Information Roadmap](http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/Shelves/ZDOCAPAR).

**Information updates on the web**

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

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

**The z/OS Basic Skills Information Center**

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

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

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

To access the z/OS Basic Skills Information Center, open your Web browser to the following Web site, which is available to all users (no login required):

How to send your comments to IBM

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

Use one of the following methods to send us your comments:
1. Send an email to mhvrncfs@us.ibm.com
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   U.S.A.
4. Fax the comments to us as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:
• Your name and address
• Your email address
• Your telephone or fax number
• The publication title and order number:
  z/OS V1R13.0 Language Environment Debugging Guide
  GA22-7560-12
• The topic and page number related to your comment
• The text of your comment.

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

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

If you have a technical problem

Do not use the feedback methods listed above. Instead, do one of the following:
• Contact your IBM service representative
• Call IBM technical support
• Visit the IBM support portal at http://www.ibm.com/systems/z/support/
Summary of Changes

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

Changes made in z/OS Version 1 Release 13

This document contains information that was previously presented in z/OS Language Environment Debugging Guide, GA22-7560-11, which supported z/OS Version 1 Release 12.

New Information:

- New fields have been added to the Common Anchor Area to support the initialization of multiple CEEPIPI environments on a single TCB and to provide enhanced I/O abend recovery support; see Table 20 on page 66 for more information.
- The output of the Language Environment VERBEXIT LEDATA output has been updated to display the contents of the USER_WORD and PAGEFRAMESIZE field; see “Understanding the Language Environment IPCS VERBEXIT LEDATA output” on page 90 for more information.
- Information about PAGEFRAMESIZE has been added to the options report produced by RPTOPTS(ON); see “Determining run-time options in effect” on page 10 for an example. This information has also been added to the dump produced by CEE3DMP; see Figure 7 on page 47 for more information.

Changed Information:

- Various CAA fields and descriptions have been updated in Table 20 on page 66.
- Information about how to generate a Language Environment U4039 abend has changed; see “Steps for generating a Language Environment U4039 abend” on page 83 for details.

Deleted information:

- As of z/OS Version 1 Release 13, IBM has withdrawn support for the Distributed Computing Environment (DCE) architecture. As a result, information related to this support has been removed.

Changes made in z/OS Version 1 Release 12

This document contains information that was previously presented in z/OS Language Environment Debugging Guide, GA22-7560-10, which supported z/OS Version 1 Release 11.

New Information:

- A new value, PLI, has been added to the COMP parameter of LEDATA. See “Understanding Language Environment IPCS VERBEXIT – LEDATA” on page 86 and “Understanding Language Environment IPCS VERBEXIT – LEDATA” on page 397 for more information.
- “Understanding the PL/I-specific LEDATA output” on page 152 has been added in Chapter 3, “Using Language Environment debugging facilities,” on page 35.
• Language Environment can now display the contents of high registers, if known, in dumps and in VERBEXIT LEDATA output.

**Changed Information:**

• The LEDATA parameters COMP(ALL) and ALL have been updated to include PL/I control block information in reports. For more information, see “Understanding Language Environment IPCS VERBEXIT – LEDATA” on page 86 and “Understanding Language Environment IPCS VERBEXIT – LEDATA” on page 397 for AMODE 64 applications.

• CAA fields and descriptions have been updated in Table 20 on page 66.

• The "Readers' Comments - We'd Like to Hear from You" section at the back of this publication has been replaced with a new section "How to send your comments to IBM" on page xix. The hardcopy mail-in form has been replaced with a page that provides information appropriate for submitting readers comments to IBM.

---

**Changes made in z/OS Version 1 Release 11**

This document contains information that was previously presented in z/OS Language Environment Debugging Guide, GA22-7560-09, which supported z/OS Version 1 Release 10.

**New information:**

• “Requesting a UNIX System Services syscall trace for debugging” on page 172 has been added in Chapter 3, “Using Language Environment debugging facilities,” on page 35.

• “Requesting a UNIX System Services syscall trace for debugging” on page 462 has been added in Chapter 12, “Using Language Environment AMODE 64 debugging facilities,” on page 377.

• “Understanding the HEAPPOOL LEDATA output” on page 118 has been added in Chapter 3, “Using Language Environment debugging facilities,” on page 35.

• “Understanding the heappool LEDATA output” on page 425 has been added in Chapter 12, “Using Language Environment AMODE 64 debugging facilities,” on page 377.

• “Using file I/O tracing to debug C/C++ file I/O problems” on page 182 has been added in Chapter 4, “Debugging C/C++ routines,” on page 175.

• “Using file I/O tracing to debug C/C++ file I/O problems” on page 470 has been added in Chapter 13, “Debugging AMODE 64 C/C++ routines,” on page 463.

**Changed information:**

• Figure 1 on page 10 has been updated in “Determining run-time options in effect” on page 10.

• Figure 11 on page 78 has been updated in “Using the machine state information block” on page 77.

• Figure 13 on page 90 has been updated in “Understanding the Language Environment IPCS VERBEXIT LEDATA output” on page 90.

• CAA field, CEECAA_CICS_EXT_REG, has been updated in Table 20 on page 66.

• Condition information for active routine has been updated in “Sections of the Language Environment dump” on page 56.
• In “Understanding Language Environment IPCS VERBEXIT – LEDATA” on page 86, the Condition Management Control Blocks section in Chapter 3, “Using Language Environment debugging facilities,” on page 35 has been updated.

• In “Understanding Language Environment IPCS VERBEXIT – LEDATA” on page 397, the Condition Management Control Blocks section in Chapter 12, “Using Language Environment AMODE 64 debugging facilities,” on page 377 has been updated.

• Figure 139 on page 363 has been updated in “Determining run-time options in effect” on page 362.
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 run-time 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 run-time debugging.

Table 3. TEST suboptions to simplify debugging

<table>
<thead>
<tr>
<th>Suboption</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>
<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>
</tbody>
</table>

```%
BLOCK4:x signed int 12
BLOCK2:x signed int 1
```

If a nonzero optimization level is used, variables do not appear in the dump.
You can use the C/C++ compiler options shown in Table 4 to make run-time debugging easier. For a detailed explanation of these options, see z/OS XL C/C++ User's Guide.

<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>
<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* for an overview and more details about Inter-procedural Analysis.

**COBOL compiler options**

When using COBOL, 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.

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 run-time debugging. For more detail on these options and functions, see the following documents:

- *Enterprise COBOL for z/OS, V4R2, Programming Guide* SC23-8529
- *Enterprise COBOL for z/OS, V3R4, Programming Guide* SC27-1412
- *COBOL for OS/390 & VM Programming Guide* SC26-9049

Table 5. COBOL compiler options for run-time debugging

<table>
<thead>
<tr>
<th>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>
<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. SYM suboption includes statement numbers in the Language Environment dump and produces a symbolic dump.</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 on page 6 to prepare your program for run-time 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 run-time debugging

<table>
<thead>
<tr>
<th>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 run-time 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>
<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 SYSTERM 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 run-time 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 PL/I for MVS & VM Programming Guide.
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 PL/I for MVS & VM Programming Guide.

Table 7. PL/I compiler options for debugging

<table>
<thead>
<tr>
<th>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 run-time 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>
<tr>
<td>LMESSAGE</td>
<td>Tells the compiler to produce run-time messages in a long form. If the cause of a run-time 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 on page 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 Enterprise PL/I for z/OS, V3R9, Programming Guide.
Table 8. Enterprise PL/I for z/OS compiler options for debugging

<table>
<thead>
<tr>
<th>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 run-time 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>
<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>
</tbody>
</table>

XREF and ATTRIBUTES | Creates a sorted cross-reference listing with attributes. |

Using Language Environment run-time options

Several run-time options affect debugging in Language Environment. The TEST run-time 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 run-time options that affect debugging. For a more detailed discussion of these run-time options, see z/OS Language Environment Programming Reference.

Table 9. Language Environment run-time options for debugging

<table>
<thead>
<tr>
<th>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>CEE DUMP</td>
<td>Specifies options to control the processing of the Language Environment® dump report.</td>
</tr>
<tr>
<td>CHECK</td>
<td>Determines if run-time checking is performed.</td>
</tr>
<tr>
<td>NODEBUG</td>
<td>Controls the COBOL USE FOR DEBUGGING declarative.</td>
</tr>
<tr>
<td>DEPTH CONDLMT</td>
<td>Specifies the limit for the depth of nested synchronous conditions in user-written condition handlers. (Asynchronous signals do not affect DEPTH CONDLMT.)</td>
</tr>
</tbody>
</table>
Table 9. Language Environment run-time options for debugging (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>INFOMSGFILTER</td>
<td>Filters user specified informational messages from the MSGFILE.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>Causes Language Environment to recognize attention interrupts.</td>
</tr>
<tr>
<td>MSGFILE</td>
<td>Specifies the ddname of the Language Environment message file.</td>
</tr>
<tr>
<td>MSGQ</td>
<td>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.</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>Produces a report that shows the run-time options in effect; see “Determining run-time options in effect” on page 10.</td>
</tr>
<tr>
<td>RPTSTG</td>
<td>Generates a report of the storage used by an enclave; see “Controlling storage allocation” on page 11.</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 run-time 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 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 run-time results can be unpredictable.</td>
</tr>
<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>XUFLOW</td>
<td>Specifies if an exponent underflow causes a routine interrupt.</td>
</tr>
</tbody>
</table>
Determining run-time options in effect

The run-time 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 run-time options, and indicates where they were set. Figure 1 shows a sample options report.

Statement of Direction

IBM intends to extend the memory architecture for large page support (1 MB pages) to support pageable large pages. Information about externals and interfaces that are related to the extension of the memory architecture is being made available in z/OS V1.13 for early planning and development purposes only.

For example, information about the PAGEFRAMESIZE run-time option, which is related to the extension of the memory architecture, is shown in the following example for planning and development purposes only. This statement of direction is subject to change or withdrawal.

---

Options Report for Enclave main 04/18/10 8:50:19 AM
Language Environment V01 R13.00

<table>
<thead>
<tr>
<th>LAST WHERE SET</th>
<th>OPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation default</td>
<td>ABPERC(NONE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ABTERMENC(ABEND)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOAIXBLD</td>
</tr>
<tr>
<td>Installation default</td>
<td>ALL31(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ANYHEAP(16384,8192,ANYWHERE,FREE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOAUTOTASK</td>
</tr>
<tr>
<td>Installation default</td>
<td>BELOWHEAP(8192,4096,FREE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CBLOPTS(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CBLOPSHPOP(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CBLOQA(OFF)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CEEDUMP(60,SYSOUT=*,FREE=END,SPIN=UNALLOC)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CHECK(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>COUNTRY(US)</td>
</tr>
<tr>
<td>Installation default</td>
<td>DEPTHCONDLMT(10)</td>
</tr>
<tr>
<td>Installation default</td>
<td>DEPTHLOC(10)</td>
</tr>
<tr>
<td>Installation default</td>
<td>DYNDUMP(*USERID,NODYNAMIC,TDUMP)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ENVAR(&quot;&quot;)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ERRCOUNT(0)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ERRUNIT(6)</td>
</tr>
<tr>
<td>Installation default</td>
<td>FILETAG(NOAUTOCVT,NOAUTOTAG)</td>
</tr>
<tr>
<td>Installation default</td>
<td>FILEISTAG(NOAUTOCVT,NOAUTOCTAG)</td>
</tr>
<tr>
<td>Default setting</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>PARM(old)</td>
<td>HEAP(4194304,5242880,ANYWHERE,KEEP,8192,4096)</td>
</tr>
<tr>
<td>Installation default</td>
<td>HEAPCHK(OFF,1,0,0,0,1024,0,1024,0)</td>
</tr>
<tr>
<td>Installation default</td>
<td>HEAPPOLS(OFF,8,10,12,10,12,10,12,256,10,1024,10,2048,10,0,10,0,10,0,0,0,0,10,0,0,10,0,0,10)</td>
</tr>
</tbody>
</table>

Figure 1. Options report example produced by run-time option RPTOPTS(ON) (Part 1 of 2)
Using the CLER CICS transaction to display and set run-time options

The CICS transaction (CLER) allows you to display all the current Language Environment run-time options for a region, and to also have the capability to modify a subset of these options. For more information about the CICS CLER transaction, see "Displaying and modifying run-time options with the CLER transaction" on page 356.

Controlling storage allocation

The following run-time options control storage allocation:
- STACK
- THREADSTACK
- LIBSTACK
- THREADHEAP
- HEAP
- ANYHEAP
- BELOWHEAP
- STORAGE
- HEAPPools
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) run-time 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 run-time options for future runs. The output is written to the Language Environment message file.

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

Figure 2 and Figure 3 on page 14 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.

---

**Figure 2.** Storage report produced by run-time option RPTSTG(ON) (Part 1 of 2)
Figure 3 on page 14 shows an example of a storage report that is produced with XPLINK.
<table>
<thead>
<tr>
<th>Stack Type</th>
<th>Initial Size</th>
<th>Increment Size</th>
<th>Maximum Used by All Concurrent Threads</th>
<th>Largest Used by Any Thread</th>
<th>Number of Segments Allocated</th>
<th>Number of Segments Freed</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACK</td>
<td>131072</td>
<td>131072</td>
<td>5416</td>
<td>5416</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>THREADSTACK</td>
<td>4096</td>
<td>4096</td>
<td>45536</td>
<td>6552</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>XPLINK STACK</td>
<td>524288</td>
<td>131072</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XPLINK THREADSTACK</td>
<td>131072</td>
<td>131072</td>
<td>20400</td>
<td></td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>LIBSTACK</td>
<td>4096</td>
<td>4096</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>THREADHEAP</td>
<td>4096</td>
<td>4096</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HEAP</td>
<td>32768</td>
<td>32768</td>
<td>286576</td>
<td>71</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3. Storage report produced by RPTSTG(ON) with XPLINK (Part 1 of 3)
**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: 16384
- Increment size: 8192
- Total heap storage used (sugg. 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 (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: 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

**HEAPPOOLS Statistics:**
- Pool 1 size: 8 Get Requests: 3
  - Successful Get Heap requests: 1-8: 3
  - Successful Get Heap requests: 9-16: 3
  - Successful Get Heap requests: 17-24: 3
  - Successful Get Heap requests: 25-32: 29

- 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 3. Storage report produced by RPTSTG(ON) with XPLINK (Part 2 of 3)*
The statistics for initial and incremental allocations of storage types that have a corresponding run-time option differ from the run-time 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

The run-time options should be tuned appropriately to avoid performance problems. See [z/OS Language Environment Programming Guide](http://www.ibm.com) 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 10. Storage report fields that display stack storage statistics

<table>
<thead>
<tr>
<th>Statistics categories</th>
<th>Field contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• STACK</td>
<td>The following fields display statistics for the upward-growing stack.</td>
</tr>
<tr>
<td>• THREADSTACK</td>
<td><strong>Initial size</strong> Actual size of the initial segment assigned to each thread. If a pthread-attributes-table is provided on the invocation of pthread-create, then the stack size specified in the pthread-attributes-table will take precedence over the STACK run-time option.</td>
</tr>
<tr>
<td>• LIBSTACK</td>
<td><strong>Increment size</strong> Size of each incremental segment acquired, as determined by the increment portion of the corresponding run-time option.</td>
</tr>
<tr>
<td>• IPT STACK</td>
<td><strong>Maximum used by all concurrent threads</strong> Maximum amount allocated in total at any one time by all concurrently executing threads.</td>
</tr>
<tr>
<td>• XPLINK STACK</td>
<td><strong>Largest used by any thread</strong> Largest amount allocated ever by any single thread.</td>
</tr>
<tr>
<td>• XPLINK THREADSTACK</td>
<td><strong>Number of segments (or increments) allocated</strong> Number of incremental segments allocated by all threads.</td>
</tr>
<tr>
<td></td>
<td><strong>Number of segments freed</strong> Number of incremental segments freed by all threads. The number of incremental segments freed could be less than the number allocated if any of the segments were not freed individually during the life of the thread, but rather were freed implicitly in the course of thread termination.</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 run-time 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 19 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>THREADHEAP</td>
<td></td>
</tr>
<tr>
<td><strong>Initial size</strong></td>
<td>Default initial allocation, as specified by the corresponding run-time option. For HEAP24, the initial size is the value of the ( \text{initsz24} ) of the HEAP option.</td>
</tr>
<tr>
<td><strong>Increment size</strong></td>
<td>Minimum incremental allocation, as specified by the corresponding run-time option. For HEAP24, the increment size is the value of the ( \text{incrsz24} ) 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>
<tr>
<td>• HEAP</td>
<td></td>
</tr>
<tr>
<td>• HEAP24</td>
<td><strong>Initial size</strong> Default initial allocation, as specified by the corresponding run-time option. For HEAP24, the increment size is the value of the ( \text{incrsz24} ) of the HEAP option.</td>
</tr>
<tr>
<td>• ANYHEAP</td>
<td><strong>Increment size</strong> Minimum incremental allocation, as specified by the corresponding run-time option. For HEAP24, the increment size is the value of the ( \text{incrsz24} ) of the HEAP option.</td>
</tr>
<tr>
<td>• BELOWHEAP</td>
<td><strong>Total heap storage used</strong> 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. 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.</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. 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.</td>
</tr>
</tbody>
</table>
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></td>
<td><strong>Successful Create Heap requests</strong></td>
</tr>
<tr>
<td></td>
<td>Number of successful Create Heap requests.</td>
</tr>
<tr>
<td></td>
<td><strong>Successful Discard Heap requests</strong></td>
</tr>
<tr>
<td></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></td>
<td><strong>Total heap storage used</strong></td>
</tr>
<tr>
<td></td>
<td>Largest total amount used by the enclave at any one time.</td>
</tr>
<tr>
<td></td>
<td><strong>Successful Get Heap requests</strong></td>
</tr>
<tr>
<td></td>
<td>Number of Get Heap requests.</td>
</tr>
<tr>
<td></td>
<td><strong>Successful Free Heap requests</strong></td>
</tr>
<tr>
<td></td>
<td>Number of Free Heap requests.</td>
</tr>
<tr>
<td></td>
<td><strong>Number of segments allocated</strong></td>
</tr>
<tr>
<td></td>
<td>Number of incremental segments allocated.</td>
</tr>
<tr>
<td></td>
<td><strong>Number of segments freed</strong></td>
</tr>
<tr>
<td></td>
<td>Number of incremental segments individually freed.</td>
</tr>
</tbody>
</table>

**HEAPPOOLS storage statistics**

The HEAPPOOLS run-time option (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 HEAPPOOLS storage statistics in the storage report, see “HEAPPOOLS storage statistics” on page 233.

**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
- Run-time options
- User-written condition handlers
- POSIX functions (used to specifically set signal actions and signal masks)

**Language Environment callable services**

Table 12 on page 21 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>Callable Service</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 run-time options

Table 13 shows the Language Environment run-time options that can affect your routine's condition handling behavior.

Table 13. Run-time options that modify condition handling

<table>
<thead>
<tr>
<th>Run-time 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 Shhh, where lhnh is a hexadecimal system abend code. User abends are specified as Udddd, where ddd 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.  
Language Environment ignores ABPERC(OCx). 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. |
| DEPTHCONDLMT    | 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'). |
Table 13. Run-time options that modify condition handling (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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 run-time option in z/OS Language Environment Programming Reference.</td>
</tr>
<tr>
<td></td>
<td>When TRAP(OFF), TRAP(OFESPIE), or TRAP(OFF,NOSPIE) is specified and either a program interrupt or abend occurs, the user exit for termination is ignored.</td>
</tr>
</tbody>
</table>
Table 13. Run-time options that modify condition handling (continued)

<table>
<thead>
<tr>
<th>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. 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 run-time 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 run-time 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 Programming 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 23 and z/OS Language Environment Programming Guide)
- POSIX signal default action (see z/OS Language Environment Programming Guide)
- The ABTERMENC run-time option (discussed below)

The ABTERMENC run-time 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_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 run-time 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 run-time messages. (For an explanation of Language Environment run-time messages, see "Interpreting run-time messages" on page 31.)

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++, 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++ run-time messages and perror() messages are directed to stderr. stderr corresponds to the ddname associated with the Language Environment MSGFILE run-time option. The destination of the printf function output can be changed by using the redirection 1>&2 at routine invocation to redirect stdout to the stderr destination. Both streams can be controlled by the MSGFILE run-time 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, run-time 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 run-time option. If the print unit is different than
the error message unit (PRTUNIT and ERRUNIT run-time 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, run-time messages are directed to the file specified in the Language Environment MSGFILE run-time 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 run-time 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 run-time 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 28).

**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 & VM, 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 27 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 z/OS Language Environment Programming Guide.</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.

---

![Figure 4. Language Environment condition token](image)

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 run-time 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 run-time messages and corrective information, see z/OS Language Environment Run-Time 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 run-time 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++ run-time 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 run-time options and callable services. (See “Controlling storage allocation” on page 11 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 run-time messages point to the nature of the error. The run-time 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 run-time 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 run-time messages” on page 31.</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 <a href="#">z/OS Language Environment Run-Time Messages</a> Check for a subsystem-generated abend or a user-specified abend.</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 <a href="#">z/OS Language Environment Run-Time Messages</a></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 run-time 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 run-time message.</td>
</tr>
<tr>
<td>Unexpected output</td>
<td>Conflicting library module names</td>
<td>See the name conflict resolution steps outlined in <a href="#">z/OS Language Environment Programming Guide</a></td>
</tr>
</tbody>
</table>
Interpreting run-time messages

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

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

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

Run-time 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 run-time 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 run-time 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 run-time 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 run-time 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 Run-Time 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 run-time 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 hhh 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 run-time 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 24, as well as z/OS Language Environment Programming Reference.

User abends
If you receive a Language Environment abend code, see z/OS Language Environment Run-Time 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 run-time option is used in combination with the TERMTHDACT run-time 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 81 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 run-time library or edcmtext can be invoked directly with the errno2 value as input.

**Format**

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
</tbody>
</table>

**Description**

The edcmtext utility displays the description and action text for C/C++ run-time 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 C/C++ run-time library is 0X'C0000000' through 0X'CFFFFFFFF'.
- 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++ run-time 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:
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++ run-time library range, the following message is displayed:

```
Notice: The errno2_value is not in the C/C++ run-time 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++ run-time 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 on Debug Tool, see:

http://www.ibm.com/software/awdtools/debugtool/

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 run-time 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

http://www.ibm.com/software/awdtools/debugtool/

You can use IBM WebSphere® Developer Debugger for System z® to get a workstation graphical interface to IBM Debug Tool for z/OS. For more information, see the following URL:


You can also use dbx to debug Language Environment applications, including C/C++ programs. 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 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)
• TERMTHDACT run-time option
• HLL-specific functions

Generating a Language Environment dump with CEE3DMP

For non-64-bit, the CEE3DMP callable service generates a dump of the run-time 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 pathname 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 describing the type, format, and destination of dump information. The options are declared as a string of keywords 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 37 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 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 42.

<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 n.</td>
</tr>
<tr>
<td>Note:</td>
<td></td>
<td>On CICS, only ENCLAVE(CURRENT) and ENCLAVE(1) settings are supported.</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 specified by the STACKFRAME option. Control blocks for files are also dumped if the FILES option was specified. See the FILES option above for more information. If the TRACE run-time 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 run-time 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>
<tr>
<td>NOSTORAGE</td>
<td>NOSTOR</td>
<td>Suppresses storage dumps.</td>
</tr>
</tbody>
</table>

Chapter 3. Using Language Environment debugging facilities  37
Table 16. CEE3DMP options (continued)

<table>
<thead>
<tr>
<th>Dump Options</th>
<th>Abbreviation</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<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 run-time 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 run-time option setting.</td>
</tr>
<tr>
<td>NOGENOPTS</td>
<td>NOGENO</td>
<td>Does not generate a run-time options report in the dump output. NOGENOPTS is the default for user-called dumps.</td>
</tr>
<tr>
<td>REGSTOR(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 REGSTOR(0) if no storage around registers is required.</td>
</tr>
</tbody>
</table>

Generating a Language Environment dump with TERMTHDACT

The TERMTHDACT run-time 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.

The TERMTHDACT suboptions QUIET, MSG, TRACE, DUMP, UAONLY, UATRACE, UADUMP, and UAIMM control the level of information available. Table 17 on page 39 lists 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>Terminal or ddname specified in MSGFILE run-time 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 run-time 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 run-time 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. <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; 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 run-time 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 run-time 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>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. <strong>Note:</strong> 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 run-time 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 UATRACE suboptions of TERMTHDACT use these dump options:
The DUMP and UADUMP suboptions of TERMTHDACT use these dump options:
- BLOCKS
- CONDITION
- ENCLAVE(ALL)
- FILES
- FNAME(CEEDUMP)
- GENOPTS
- NOBLOCKS
- NOENTRY
- NOSTORAGE
- STACKFRAME(ALL)
- 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 run-time option, see the 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/1 Considerations**
  After a normal return from a PL/1 ERROR ON-unit, or from a PL/1 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/1 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 run-time 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>
<tr>
<td>TRACE</td>
<td>The traceback is written to the CESE queue, followed by U4038 abend with nodump option.</td>
<td>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). U4038 abend with nodump option. Message to CESE or MSGFILE.</td>
</tr>
<tr>
<td>DUMP</td>
<td>CCE03DUMP to CESE queue followed by U4038 abend with nodump option.</td>
<td>CCE03DUMP to new transaction server queue which CICS will read and write to CICS transaction dump later. U4038 abend with nodump option. Message to CESE or MSGFILE.</td>
</tr>
<tr>
<td>UATRACE</td>
<td>U4039 abend with traceback to CESE queue followed by U4038 abend with nodump option.</td>
<td>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). U4038 abend with nodump option. Message to CESE or MSGFILE.</td>
</tr>
</tbody>
</table>

Chapter 3. Using Language Environment debugging facilities 41
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>
| 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 run-time 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 192.

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 271.

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 292.

Understanding the Language Environment dump

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

Figure 7 on page 47 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 56.

The CEE3DMP was generated by the C program CELSAMP shown in Figure 5 on page 43. CELSAMP uses the DLL CELDLL shown in Figure 6 on page 46.
#pragma options(SERVICE("1.1.d"), NOOPT, TEST(SYM))
#pragma runopts(TERMTHDACT(UADUMP), POSIX(ON), DYNDUMP(DYNAMIC,))
#pragma runopts(TRACE(ON, IM, NODUMP, LE=1), HEAPCHK(ON, 1, 10, 10))
#pragma runopts(RPTSTG(ON), HEAPPOOLS(ON))
define_OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <dll.h>
#include <signal.h>
#include <leawi.h>
#include <ceedcct.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) {
    /* values for handling the conditions */
define percolate 20
    printf(">>> Thread_CleanUp: Msg # is %d\n",cond->tok_msgno);
    if (!threads_joined) {
        printf(">>> Thread_CleanUp: Unlocking mutex\n");
        pthread_mutex_unlock(&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;
    }
    *result = percolate;
    printf(">>> Thread_CleanUp: Percolating condition\n");
}

/*********************************************************************/
/* 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);
}

Figure 5. The C program CELSAMP (Part 1 of 3)
main()
{
    dllhandle * handle;
    int    i = 0;
    FILE*  fp1;
    FILE*  fp2;
    _FEEDBACK fc;
    _INT4 token;
    _ENTRY pgmptr;

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

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

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

    printf("Create 2nd thread\n");
    if (pthread_create(&thread[0],NULL,thread_func,(void *)t2) == -1) {
      perror("Could not create thread #2");
      exit(104);
    }

    printf("Register thread cleanup condition handler\n");
    pgmptr.address = (_POINTER)thread_cleanup;
    pgmptr.nesting = NULL;
    token = 1;
    CEEHDLR (&pgmptr, &token, &fc);
    if ( _FBCHECK ( fc , CEE000 ) !=0)
    { 
      printf(" CEEHDLR failed with message number %d\n",fc.tok_msgno); 
      exit(105); 
    }

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

    printf("Query DLL with incorrect function name\n");
    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\n");
    pgmptr.address = (_POINTER)dllqueryfn(handle,"dump_n_perc");
    if (pgmptr.address == NULL) {
      perror("Could not find dump_n_perc");
      exit(107);
    }
}

Figure 5. The C program CELSAMP (Part 2 of 3)
printf("Register condition handler...
");
pmptr.nesting = NULL;
token = 2;
CEEHDLR (&pgmptr, &token, &fc);
if ( FCHECK (fc, CEE000) != 0 ) {
    printf("CEEHDLR failed with message number %d\n",
           fc.tok_msgno);
    exit(100);
}
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 5. The C program CELSAMP (Part 3 of 3)
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 56.

**Statement of Direction**

IBM intends to extend the memory architecture for large page support (1 MB pages) to support pageable large pages. Information about externals and interfaces that are related to the extension of the memory architecture is being made available in z/OS V1.13 for early planning and development purposes only.

For example, information about the PAGEFRAMESIZE run-time option, which is related to the extension of the memory architecture, is shown in the following example for planning and development purposes only. This statement of direction is subject to change or withdrawal.

```c
/* DLL containing Condition Handler that takes dump and percolates */
#pragma options(SERVICE("1.3.b"),TEST(SYM),NOOPT)
#pragma export(dump_n_perc)
#include <stdio.h>
#include <leawi.h>
#include <stdlib.h>
#include <string.h>
#include <ceeedcct.h>
char wsa_array[10] = { 'C','E','L','D','L','L',' ','W','S','A'};
#define OPT_STR "THREAD(ALL) BLOCKS STORAGE GENOPTS"
#define TITLE_STR "Sample dump produced by calling CEE3DMP"

void dump_n_perc(_FEEDBACK *cond, _INT4 *input_token,
                 _INT4 *result, _FEEDBACK *new_cond) {
    /* values for handling the conditions */
#define percolate 20

    _CHAR80 title;
    _CHAR255 options;
    _FEEDBACK fc;

    printf(">>> dump_n_perc: Msg # is %d\n",cond->tok_msgno);

    /* check if the DIVIDE-BY-ZERO message (0C9) */
    if (cond->tok_msgno == 3209) {
        memset(options,' ',sizeof(options));
        memcpy(options,OPT_STR,sizeof(OPT_STR)-1);
        memset(title,' ',sizeof(title));
        memcpy(title,TITLE_STR,sizeof(TITLE_STR)-1);
        printf(">>> dump_n_perc: Taking dump\n");
        CEE3DMP(title,options,&fc);
        if ( _FBCHECK ( fc , CEE000 ) !=0 ) {
            printf("CEE3DMP failed with msgno %d\n",fc.tok_msgno);
            exit(299);
        }
    }
    *result = percolate;
    printf(">>> dump_n_perc: Percolating condition\n");
}
```

Figure 6. The C DLL CELDLL
Figure 7. Example dump using CEE3DMP (Part 1 of 10)
Fully Qualified Names
DSA Entry Program Unit Load Module
1 dump_n_perc //POSIX.CRTL.C(CELDLL)' CELDLL
4 main //POSIX.CRTL.C(CELSAMP)' CELSAMP

[8] Condition Information for Active Routines

Condition Information for //POSIX.CRTL.C(CELSAMP)' (DSA address 265E9208)
CIB Address: 265E5A08
Current Condition:
CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).
Location:
Program Unit: //POSIX.CRTL.C(CELSAMP)' Entry: main Statement: 150 Offset: +000009BA
Machine State:
ILC..... 0002 Interruption Code..... 0009
PSW..... 07B05A00 A5E0A07C
GPR0..... 00000000_00000000 GPR1..... 00000000_00000000 GPR2..... 00000000_00000000 GPR3..... 00000000_00000000
GPR4..... 00000000_00000000 GPR5..... 00000000_00000000 GPR6..... 00000000_00000000 GPR7..... 00000000_00000000
GPR8..... 00000000_00000000 GPR9..... 00000000_00000000 GPR10.... 00000000_00000000 GPR11.... 00000000_00000000
GPR12.... 00000000_00000000 GPR13.... 00000000_00000000 GPR14.... 00000000_00000000 GPR15.... 00000000_00000000

Storage dump near condition, beginning at location: 25E0A06A
+000000 25E0A06A 4400C1AC 5800D0AC 41600001 8E600020 1D601807 5000D0AC 4400C1AC 58F039E2 |..A......-...-..&.....A..0.S|
GPREG STORAGE:
Storage around GPR0 (00000000)
+0020 00000000 Inaccessible storage.
+0020 00000000 Inaccessible storage.
+0040 00000000 Inaccessible storage.

Parameter, Registers, and Variables for Active Routines:
dump_n_perc (DSA address 265ECE90):
UPSTACK DSA
Parameters:
new_cond struct _FEEDBACK *
0x25E0E5C
result signed long int *
0x265E6C4
input_token signed long int *
0x265E6B8
cond struct _FEEDBACK *
0x265E5F0
Saved Registers:
GPR0..... 265E6D88 GPR1..... 265ECE98 GPR2..... 265ECCB8 GPR3..... 269A90FA
GPR4..... 265ECCB8 GPR5..... 269A90FA GPR6..... 00000000 GPR7..... 25E0E410
GPR8..... A5ED682 GPR9..... 265E6148 GPR10.... 265EACDF GPR11.... A5F97290
GPR12.... 265E6B48 GPR13.... 265ECC90 GPR14.... A699A1F6 GPR15.... A5EF8428
GPREG STORAGE:
Storage around GPR0 (25E0E688)
-0020 25E0E688 40404040 40404040 40404040 40404040 40404040 40404040 40404040 | .|
.
.
.

Local Variables:
title[0..6] unsigned char 'S' 'a' 'm' 'p' 'l' 'e'

title[7..13] 'd' 'u' 'p' 'o' 'd' 'u' 'c' 'e' 

title[14..20] 'b' 'y' 'C' 'E' 'E'

title[21..27] 'C' 'E'

title[28..34] 'C' 'E'

title[35..41] 'C' 'E'

title[42..48] 'C' 'E'

title[49..55] to title[76..76] elements same as above.

title[77..79] 'C' 'E'

title[80..86] 'C' 'E'

title[87..93] 'C' 'E'

title[94..100] 'C' 'E'

Figure 7. Example dump using CEE3DMP (Part 2 of 10)
Figure 7. Example dump using CEE3DMP (Part 3 of 10)
Figure 7. Example dump using CEE3DMP (Part 4 of 10)
Figure 7. Example dump using CEE3DMP (Part 5 of 10)
<table>
<thead>
<tr>
<th>Control Blocks for Active Routines:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSA for CEEOPML2: 26938090</td>
</tr>
<tr>
<td>+000000  FLAGS... 0000 member... CCC  BKC... 26937DE0  FMC..... 26938210  R14..... A5F5E7F6</td>
</tr>
<tr>
<td>+000110  R15..... A5F60090  R0..... 25F5F2D4  R1...... 26938114  R2...... 2671C43C  R3...... 2693D708</td>
</tr>
<tr>
<td>+000224  R4....... 00000000  R5....... 2660A954  R6...... 265E61A0  R7........ 2660A930  R8....... 25E00ED0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Storage for Active Routines:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSA frame: 26937DE0</td>
</tr>
<tr>
<td>+000000  26937DE0 10CCCCCC 26937D38 26938090 A6411BDE A5F5E148 25F5F2D4 25E0F158 265E61A0</td>
</tr>
<tr>
<td>+000020  26937E00 25E00DC2 26937D38 25E00ED0 269378A0 26613128 00000000 00000080 CCCCCCCC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control Blocks Associated with the Thread:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAA: 26936BD8</td>
</tr>
<tr>
<td>+000000  26936BD8 00000800 00000000 26937D20 00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+000020  26936BF8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enclave variables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>*.C(CELSAMP)'&gt;mut</td>
</tr>
<tr>
<td>struct __m unsigned long int</td>
</tr>
<tr>
<td>+000000  643868976</td>
</tr>
<tr>
<td>*.C(CELSAMP)'&gt;thread[0]</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>+000000  \x26' \x25' 'p' '0' '0' '0' '0'</td>
</tr>
<tr>
<td>*.C(CELSAMP)'&gt;thread[1]</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>+000000  \x26' \x25' 'y' '0' '0' '0' '0'</td>
</tr>
<tr>
<td>*.C(CELSAMP)'&gt;threads Joined</td>
</tr>
<tr>
<td>signed int 0</td>
</tr>
<tr>
<td>*.C(CELSAMP)'&gt;thread_func</td>
</tr>
<tr>
<td>void * () 0x25E00D88</td>
</tr>
<tr>
<td>*.C(CELSAMP)'&gt;thread_cleanup</td>
</tr>
<tr>
<td>void () 0x25E00B28</td>
</tr>
<tr>
<td>*.C(CELDLL)'&gt;wsa_array[0..6]</td>
</tr>
<tr>
<td>unsigned char 'C' 'E' 'L' 'D' 'L' 'L' ' '</td>
</tr>
<tr>
<td>*.C(CELDLL)'&gt;wsa_array[7..9]</td>
</tr>
<tr>
<td>unsigned char 'W' 'S' 'A'</td>
</tr>
<tr>
<td>*.C(CELDLL)'&gt;dump_n_perc</td>
</tr>
<tr>
<td>void () 0x2699A0C0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enclave Control Blocks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDB: 25E15780</td>
</tr>
<tr>
<td>+000000  25E15780 35C5C5C5 C4C24040 C7000001 25E16A08 25E15E88 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+000020  25E15700 25E15CO 25E15CF0 A5E1B000 25E15300 00000000 00000000 25E15B00 00000000 .*.+0V..+j....</td>
</tr>
<tr>
<td>+000040  25E15700 00000000 00000000 00000100 00000000 00000000 00000000 00000000 00000000 00000000 .*.+Vm...D/..</td>
</tr>
<tr>
<td>+000060  25E15810 00000560 2671C000 25E13FC0 00000100 25E177F0 00000000 26008038 25E16848 .*.+Vv...v....</td>
</tr>
<tr>
<td>+000080  25E15830 00000000 00000B8E 00000000 00000000 00000003 00000000 00000600 008FF07B .*.+A0.00+00.25E15850</td>
</tr>
<tr>
<td>+0000A0  25E15850 00000001 00000100 25E0CA0 25E15858 00000000 00000000 00000000 00000003 00000000 .*.+B0.00+01.25E16A68</td>
</tr>
<tr>
<td>+000000  25E16A08 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .*.+ZQ...ZQ.00.25E9F08</td>
</tr>
<tr>
<td>+000020  25E16A28 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .*.+Q...W..ZQ.00.25E9F08</td>
</tr>
<tr>
<td>+000040  25E16A48 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 .*.+Q...W..ZQ.00.25E9F08</td>
</tr>
<tr>
<td>+000060  25E16A68 .+00011F 25E16B27 same as above</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutex and Condition Variable Blocks (MCVB+MHT+CHT): 2671C018</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000  2671C018 000085F0 2671C044 00003F38 00001FC0 00000000 265E8500 2671C44 0000000F8 .*.+B...+e..D..8</td>
</tr>
<tr>
<td>+000020  2671C038 000007C0 00000000 265E85E8 00000000 00000000 00000000 00000000 00000000 .*.+eY...+.e..D.c100.2671C058</td>
</tr>
</tbody>
</table>

Figure 7. Example dump using CEE3DMP (Part 6 of 10)
Figure 7. Example dump using CEE3DMP (Part 7 of 10)
<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 15.25.14.872201 Date 2010.04.07 Thread ID... 2625860000000000</td>
<td>main</td>
</tr>
<tr>
<td>+000010</td>
<td>Member ID.... 03 Flags...... 000000 Entry Type...... 00000001</td>
<td></td>
</tr>
<tr>
<td>+000020</td>
<td>94818995 40404040 40404040 40404040 40404040 40404040 40404040 40404040</td>
<td>main</td>
</tr>
<tr>
<td>+000030</td>
<td>6066E40 000F8F550 40979998 95A3864D 5D404040 40404040 40404040 40404040</td>
<td>--&gt;(085) printf()</td>
</tr>
<tr>
<td>+000040</td>
<td>6066E40 000F8F550 40979998 95A3864D 5D404040 40404040 40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000050</td>
<td>6066E40 000F8F550 40979998 95A3864D 5D404040 40404040 40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000060</td>
<td>6066E40 000F8F550 40979998 95A3864D 5D404040 40404040 40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000070</td>
<td>6066E40 000F8F550 40979998 95A3864D 5D404040 40404040 40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000080</td>
<td>Time 15.25.14.998940 Date 2010.04.07 Thread ID... 2625860000000000</td>
<td>&lt;--(085) R15=0000001E ERRNO=0000</td>
</tr>
<tr>
<td>+000090</td>
<td>Member ID.... 03 Flags...... 000000 Entry Type...... 00000002</td>
<td></td>
</tr>
<tr>
<td>+0000A0</td>
<td>4C60604D F0F8F55D 40D9F1F5 7EF0F0F0 F0F0F0F1 C440C5D9 D9D5D67E F0F0F0F0</td>
<td>&lt;--(085) R15=0000001D ERRNO=0000</td>
</tr>
<tr>
<td>+0000B0</td>
<td>F0F0F0F0 00000000 00000000 00000000 00000000 00000000 00000000 00000000</td>
<td>0000.00000000000000000000</td>
</tr>
<tr>
<td>+0000C0</td>
<td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+0000D0</td>
<td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+0000E0</td>
<td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+0000F0</td>
<td>00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000</td>
<td></td>
</tr>
</tbody>
</table>

[15]Enclave Storage:

<table>
<thead>
<tr>
<th>Initial (User) Heap: 26609018</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 26609018 C8C1D5C3 25E15C90 25E15C90 00000000 A6609018 2660E260 00008000 00002DB8</td>
</tr>
<tr>
<td>+000020 26609038 26609018 00001030 D2C4C240 00000000 00000005 00000005 2660A070 2660A258</td>
</tr>
<tr>
<td>+000040 26609058 E3E5C240 00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+000060 26609078 00000000 2660A440 2660A47D 2660A48A 2660A4F7 2660A534 2660A571 2660A5AE</td>
</tr>
<tr>
<td>+000080 2660A988 00000000 2660A440 2660A47D 2660A48A 2660A4F7 2660A534 2660A571 2660A5AE</td>
</tr>
</tbody>
</table>

Figure 7. Example dump using CEE3DMP (Part 8 of 10)
[16] File Status and Attributes:

[17] Run-Time Options Report:

<table>
<thead>
<tr>
<th>LAST WHERE SET</th>
<th>OPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation default</td>
<td>ABPERC(NONE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ABTERMENC(ABEND)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOAIXBDL</td>
</tr>
<tr>
<td>Installation default</td>
<td>ALL(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ANYHEAP(16384,B192,ANYWHERE,FREE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOAUTOTASK</td>
</tr>
<tr>
<td>Installation default</td>
<td>BELLOWHEAP(B192,4096,FREE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CBLDPTS(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CBLPSHPOP(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CBLODA(OF)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>CEEILMD(0,SYSDT=*,FREE=END,SPIN=UNALLLOC)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CHECK(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>COUNTRY(US)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NODEBUG</td>
</tr>
<tr>
<td>Installation default</td>
<td>DEPTHCONDLMT(10)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>DUMP(POSIX,DEBUG,HLE7770,DYNAMIC)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ENVAR(***)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ERRCOUNT(0)</td>
</tr>
<tr>
<td>Installation default</td>
<td>ERRUNIT(6)</td>
</tr>
<tr>
<td>Installation default</td>
<td>FILEHIST</td>
</tr>
<tr>
<td>Installation default</td>
<td>FILETAG(NOAUTOCVT,NOAUTOTAG)</td>
</tr>
<tr>
<td>Default setting</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>Installation default</td>
<td>HEAP(32768,32768,ANYWHERE,B192,4096)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>HEAPCHK(ON,1,0,10,10,1024,0,1024,0)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>HEAPPOLS(ON,8,10,32,10,128,10,256,10,1024,10,2048,10,0,10,0,10,0,10,0,10,0,10,0,10,0)</td>
</tr>
<tr>
<td>Installation default</td>
<td>INFOMSGFILTER(ON,10)</td>
</tr>
<tr>
<td>Installation default</td>
<td>INPCMP</td>
</tr>
<tr>
<td>Installation default</td>
<td>INTERRUPT(OF)</td>
</tr>
<tr>
<td>Installation default</td>
<td>LSTACK(4096,4096,FREE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>MSGFILE(SYSDT,FBA,121,0,NOENQ)</td>
</tr>
<tr>
<td>Installation default</td>
<td>MSGQ(15)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>Ignored</td>
<td>NONONITSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>Installation default</td>
<td>OCSTATUS</td>
</tr>
<tr>
<td>Installation default</td>
<td>PAGERAMESIZE(4K,4K,4K)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOPC</td>
</tr>
<tr>
<td>Installation default</td>
<td>PLITASKCOUNT(20)</td>
</tr>
<tr>
<td>Programmer default</td>
<td>POSIX(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>PROFILE(OF,OFF)</td>
</tr>
<tr>
<td>Installation default</td>
<td>PRUNIT(6)</td>
</tr>
<tr>
<td>Installation default</td>
<td>PUBUNIT(OF)</td>
</tr>
<tr>
<td>Installation default</td>
<td>RINIT(15)</td>
</tr>
<tr>
<td>Installation default</td>
<td>RPTOPTS(ON)</td>
</tr>
</tbody>
</table>

Figure 7. Example dump using CEE3DMP (Part 9 of 10)
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 204.
- For COBOL routines, see “Finding COBOL information in a dump” on page 251.
- For Fortran routines, see “Finding Fortran information in a Language Environment dump” on page 276.
- For PL/I routines, see “Finding PL/I for MVS & VM information in a dump” on page 295.
### Table 19. Contents of the Language Environment dump

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [1] Page Heading           | The page heading section appears on the top of each page of the dump and contains the following information:  
  - CEE3DMP identifier  
  - Title: For dumps generated as a result of an unhandled condition, the title is “Condition processing resulted in the Unhandled condition.”  
  - Product abbreviation of Language Environment  
  - Version number  
  - Release number  
  - Date  
  - Time  
  - Page number  

The contents of the second line of the page heading vary depending on the environment in which the CEEDUMP is issued.  

For CEEDUMPs produced under a batch environment, the following items are displayed:  
  - ASID: Describes the address space ID.  
  - Job ID: Describes the JES Job ID.  
  - Job name: Describes the job name.  
  - Step name: Describes the job’s step name in which the CEEDUMP was produced.  
  - UserID: Describes the TSO userid who issued the job.  

For jobs running with POSIX(ON), the following additional items are displayed:  
  - PID: Displays the associated process ID.  
  - Parent PID: Displays the associated parent PID.  

For CEEDUMPs produced under the z/OS UNIX shell, the following items are displayed:  
  - ASID: Describes the address space ID.  
  - PID: Displays the associated process ID.  
  - Parent PID: Displays the associated parent PID.  
  - User name: Contains the user ID associated to the CEEDUMP.  

For CEEDUMPs produced under CICS, the following items are displayed:  
  - Transaction ID and task number.  

| [2] CEE3845I CEEDUMP Processing started. | 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. |
| [3] Caller Program Unit and Offset | Identifies the routine name and offset in the calling routine of the call to the dump service |
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 run-time 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>• 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>
<tr>
<td>[5] - [17] Enclave Information</td>
<td>These sections show information that is specific to an enclave. When multiple enclaves are dumped, these sections will appear for each enclave.</td>
</tr>
<tr>
<td>[5] Enclave Identifier</td>
<td>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 &quot;Multiple enclave dumps&quot; on page 78.</td>
</tr>
<tr>
<td>[6] - [12] 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>[6] Information for thread</td>
<td>Shows the system identifier for the thread. Each thread has a unique identifier.</td>
</tr>
<tr>
<td>Section Number and Heading</td>
<td>Contents</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------</td>
</tr>
<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 CEEPPA 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>• 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>
</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 C/C++ 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 C, use TEST(SYM).  
    - For 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, use TEST(SYM)  
    - For Fortran, use SDUMP.  
    - For PL/I, arguments and variables are not displayed. |

| [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 |
<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[11] Storage for Active Routines</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>[12] Control Blocks Associated with the Thread</strong></td>
<td>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 78.</td>
</tr>
<tr>
<td><strong>[13] Enclave variables:</strong></td>
<td>Displays language specific global variables. This section also shows the variable type. Variables are displayed only if the symbol tables are available.</td>
</tr>
</tbody>
</table>
| **[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 run-time options are set.  
  • If the POSIX run-time 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, writable static area (WSA) address, and the thread id of the thread that loaded the DLL.  
  • If the HEAPCHK run-time 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 run-time 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 run-time 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] Run-Time Options Report** | Lists the Language Environment run-time 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 run-time option is set to ON, the process level latch table. Other language-specific control blocks can appear in this section. |
| **[19] Additional Language Specific Information** | 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. |
| **[20] CEE3846I CEEDUMP Processing completed.** | 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. |
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 in Figure 7 on page 47.

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 run-time 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 8 on page 64 shows the format of the upward-growing stack frame.

Note: The Member-defined fields are reserved for the specific higher level language.
Downward-growing (XPLINK) stack frame section

Figure 9 on page 65 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 66 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.
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>CEEAAFLAG0</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    CEECAAXHDL. A flag used by the condition handler. If the flag is set to 1, the application requires immediate return/percolation to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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    CEECAATHFN. 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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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 run-time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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>15C</td>
<td>Address</td>
<td>4</td>
<td>CEECAHLLEXIT</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>CEECAHOOK</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>(debugger entry)</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>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>CEECAALLOC</td>
<td>ALLOCATE descr. built</td>
</tr>
<tr>
<td>1AC</td>
<td>Char</td>
<td>4</td>
<td>CEECAASTATE</td>
<td>New statement begins</td>
</tr>
<tr>
<td>1B0</td>
<td>Char</td>
<td>4</td>
<td>CEECAENTRY</td>
<td>Block entry</td>
</tr>
<tr>
<td>1B4</td>
<td>Char</td>
<td>4</td>
<td>CEECAEXIT</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>CEECAIFTRUE</td>
<td>True part of IF</td>
</tr>
<tr>
<td>1D0</td>
<td>Char</td>
<td>4</td>
<td>CEECAIFFALSE</td>
<td>False part of IF</td>
</tr>
<tr>
<td>1D4</td>
<td>Char</td>
<td>4</td>
<td>CEECAWHEN</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>CEECAARSVHD1</td>
<td>Reserved hook</td>
</tr>
<tr>
<td>1E4</td>
<td>Char</td>
<td>4</td>
<td>CEECAARSVHD2</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>CEECAAMEVMASK</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>CEECAACRTL</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>CEECAAEDCV</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>
<tr>
<td>21F</td>
<td>Char</td>
<td>1</td>
<td>CEECAASP_FLAGS2</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>CEECAAFCMUTEXOK</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>CEECAEDCOV</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>CEECAATRTSPACE</td>
<td>C/370 Open Libvec</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>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_TCASRV_USERWORD</td>
<td>TCA Service Rtn Vctr</td>
</tr>
<tr>
<td>25C</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASRV_WORKAREA</td>
<td></td>
</tr>
<tr>
<td>260</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASRV_GETMAIN</td>
<td></td>
</tr>
<tr>
<td>264</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASRV_FREEMAIN</td>
<td></td>
</tr>
<tr>
<td>268</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASRV_LOAD</td>
<td></td>
</tr>
<tr>
<td>26C</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASRV_DELETE</td>
<td></td>
</tr>
<tr>
<td>270</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASRV_EXCEPTION</td>
<td></td>
</tr>
<tr>
<td>274</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASRV_ATTENTION</td>
<td></td>
</tr>
<tr>
<td>278</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_TCASRV_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 (6)</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>
<tr>
<td>2AE</td>
<td>Bit (6)</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>
</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>2AF</td>
<td>Bit</td>
<td>1</td>
<td>CEECAAFLAG2</td>
<td>CAA Flag 2, defined as follows: 0: Bimodal addressing is available. 1: Vector hardware is available. 2: Thread terminating. 3: Initial thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4: Library trace is active. The TRACE run-time option was set. 5: Reserved 6: CEECAA_ENQ_Wait_Interruptible. Thread is in an enqueue wait.</td>
</tr>
<tr>
<td>2B0</td>
<td>Unsign</td>
<td>1</td>
<td>CEECAALEVEL</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</td>
<td>1</td>
<td>CEECAA_PM</td>
<td>Image of current program mask.</td>
</tr>
<tr>
<td>2B2</td>
<td>Bit</td>
<td>16</td>
<td>CEECAA_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>CEECAAGETLS</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>CEECALBOS</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>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>CEECAADM C</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>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>2D8</td>
<td>Address</td>
<td>4</td>
<td>CEEAAERR</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>CEECAADDSA</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>CEECAASECTSIZ</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>CEECAASSEXPNT</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>CEECAAEYEPTR</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>
<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>CEECAAGETSI</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>CEECAAFRGL</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>4–7</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>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>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>CEECAAOESS</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>
<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>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>
</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>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_sigsynchsflag was on last time CEEOSIGG 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_PUTB. 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>This field is the thread identifier</td>
</tr>
<tr>
<td>33C</td>
<td>Char</td>
<td>8</td>
<td>CEECAATHDID</td>
<td>Reserved</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_CTOC</td>
<td>TOC anchor for CRENT.</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>CEECAAMEMBR</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>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>CEECAAEDCHPXV</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>
</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>37C</td>
<td>Signed</td>
<td>4</td>
<td>CEECAA_SYS_RTNCODE</td>
<td>System (kernel) return code.</td>
</tr>
<tr>
<td>380</td>
<td>Signed</td>
<td>4</td>
<td>CEECAA_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>CEECAA_JIT1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>38C</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_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>CEECAA_FORDBG</td>
<td>Ptr to AFHDBHIM - FORTRAN hook interface.</td>
</tr>
<tr>
<td>39C</td>
<td>Bit (8)</td>
<td>1</td>
<td>CEECAAAB_STATUS</td>
<td>Validity flags.</td>
</tr>
<tr>
<td>39D</td>
<td>Unsigned</td>
<td>1</td>
<td>CEECAA_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>CEECAAAB_GR0</td>
<td>Reg 0 at the time of abend.</td>
</tr>
<tr>
<td>3A4</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAAB_ICD1</td>
<td>SDWAICD1.</td>
</tr>
<tr>
<td>3A8</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAAB_ABCC</td>
<td>SDWAABCC.</td>
</tr>
<tr>
<td>3AC</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAAB_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>CEECAA_LERSN1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>3B8</td>
<td>Address</td>
<td>4</td>
<td>CEECAAHERP</td>
<td>Address of CEEHERP routine.</td>
</tr>
<tr>
<td>3BC</td>
<td>Address</td>
<td>4</td>
<td>CEECAAUSTKBOSS</td>
<td>Start of user stack segment.</td>
</tr>
<tr>
<td>3C0</td>
<td>Address</td>
<td>4</td>
<td>CEECAAUSTKEOS</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>CEECAACEL4CEEDLLF</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>
<tr>
<td>3F0</td>
<td>Char</td>
<td>4</td>
<td>CEECA_USER_WORD</td>
<td>4-byte user field available for application use. In pre-initialization (CEEPPI) environments, this field is initialized in the IPT CAA from the CEEPPI set_user_word function. This field is initialized to 0 in non-CEEPPI environments (including all nested enclaves), and for all non-IPT CAAs in CEEPPI environments. This field is not otherwise accessed by Language Environment.</td>
</tr>
<tr>
<td>3F4</td>
<td>Address</td>
<td>4</td>
<td>CEECA_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>
</tbody>
</table>
**Condition information block**

Figure 10 on page 73 shows the condition information block. The Language Environment condition manager creates a condition information block (CIB) for each condition 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 located 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 leawi.h.
Figure 10. Condition information block (Part 1 of 2)
The flags for Condition Flag 4:

- 2  The resume cursor has been moved
- 4  Message service has processed the condition
- 8  The resume cursor has been moved explicitly

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
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

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 11 on page 78 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 90 for the contents of CEEDLLF fields.

Multiple enclave dumps

Figure 12 on page 80 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.
Figure 12. Language Environment dump of multiple enclaves
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 run-time 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 run-time 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 38.

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(abc ode)
The ABPERC run-time 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 run-time environments. The following sections describe the recommended steps needed to generate a system dump in a batch, IMS, CICS, and z/OS UNIX shell run-time environments. Other methods may exist, but these are the recommended steps for generating a system dump.

For details on setting Language Environment run-time options, see z/OS Language Environment Programming Guide.

Steps for generating a system dump in a batch run-time environment

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

1. Specify run-time 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 38.

2. Decide whether to include a SYSMDUMP DD card or use the DYNDUMP
run-time 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 run-time option with the following information:

     DYNDUMP (hlq,DYNAMIC,TDUMP)

3. Rerun the program.

Steps for generating a system dump in an IMS run-time
environment

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

1. Specify run-time 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 38

2. Decide whether to include a SYSMDUMP DD card or use the DYNDUMP
run-time 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 run-time option with the following information:

     DYNDUMP (hlq,DYNAMIC,TDUMP)

3. Rerun the program.

Steps for generating a system dump in a CICS run-time
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 run-time 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 run-time environment. When you are done, you will have generated a system dump in a CICS run-time environment.

1. Specify run-time 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 38.

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

   ```
   CEMT SET TRD(4088) 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 run-time options, a Language Environment U4039 abend occurs which generates a system dump.

1. Specify DUMP=YES in CICS DFHSIT.
2. Specify run-time 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 run-time options, where suboption is UAONLY, UADUMP, UATRACE, or UAIMM.

```plaintext
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 38.

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 run-time options:

```plaintext
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 38.

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 run-time 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 CEEIPCS 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 SYSLPARMLIB 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(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
- Run-time Options
- Storage Management Control Blocks
- Condition Management Control Blocks
- Message Handler Control Blocks
- C/C++ Control Blocks
- COBOL Control Blocks

Format

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

Report Type Parameters:
- SUM
- HEAP | STACK | SM
- HPT(number) | HPTTCB(address) | HPTCELL(address) | HPTLOC(location)
- CM
- MH
- CEDUMP
- 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
Run-time 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 heappools 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 poolid is formatted. If the value is a single number (1-12), the trace for the specific heappools poolid 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 poolid.

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

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

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

31  Display entries located in virtual storage below the bar.
64  Display entries located in virtual storage above the bar.
ALL  Display entries located in virtual storage below or above the bar.

Notes:
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.

**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.

**AST**
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++ Run-Time 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 112.

**EXC**
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(\text{ca\text{-}a\text{-}address})

specifies the address of the CAA. If not specified, the CAA address is obtained from the TCB.

DSA(\text{d\text{-}a\text{-}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(\text{t\text{-}c\text{-}b\text{-}address})

specifies the address of the TCB. If not specified, the TCB address of the current TCB from the CVT is used.

ASID(\text{a\text{-}s\text{-}i\text{d}}\text{\text{-}address})

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(\text{v\text{-}a\text{l\text{-}u\text{e}}})

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 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" on page 90.
Understanding the Language Environment IPCS VERBEXIT

LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of the Language Environment run-time environment control blocks from a system dump. Figure 13 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) run-time option when running the program CELSAMP in Figure 5 on page 43.

“Sections of the Language Environment LEDATA VERBEXIT formatted output” on page 107 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.

Statement of direction

IBM intends to extend the memory architecture for large page support (1 MB pages) to support pageable large pages. Information about externals and interfaces that are related to the extension of the memory architecture is being made available in z/OS V1.13 for early planning and development purposes only. IBM intends to extend the memory architecture for large page support (1 MB pages) to support pageable large pages.

For example, information about the PAGEFRAMESIZE run-time option, which is related to the extension of the memory architecture, is shown in the following example for planning and development purposes only. This statement of direction is subject to change or withdrawal.

---

Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 1 of 18)
Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 2 of 18)
Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 3 of 18)
<table>
<thead>
<tr>
<th>LAST WHERE SET</th>
<th>Override OPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>ABTERMENC(ABEND)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>NOAIXBD</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>ALL31(ON)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>ANYHEAP(0000016384,0000008192,ANY,FREE)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>NOAUTOTASK</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>BELOWHEAP(00008192,00004096,FREE)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>CBLOPTS(ON)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>CBLSHPOP(ON)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>CBLODA(OFF)</td>
</tr>
<tr>
<td>DD:CEEOPTS OVR</td>
<td>CEEDUMP(0000000000,SYSOUT=*,FREE=END,SPIN=UNALLOC)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>CHECK(ON)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>COUNTRY(US)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>DEPTHCONDLMT(0000000010)</td>
</tr>
<tr>
<td>DD:CEEOPTS OVR</td>
<td>DYNDUMP(POSIX.DEBUGG,HLE7780,DYNAMIC,TDUMP)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>ENVAR(&quot;&quot;)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>ERRCOUNT(0000000000)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>ERRUNIT(000000006)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>FILELIST</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>FILETAG(NOAUTOCVT,NOAUTOTAG)</td>
</tr>
<tr>
<td>DEFAULT SETTING OVR</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>HEAP(0000032768,0000032768,FREE)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>HEAPCHK(ON,0000000001,0000000000,0000000010,0000000010,00001024,00,00001024,00)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>HEAPPOOLS(ON,00000008,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010,00000010)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>INFOMSGFILTER(OFF)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>INQPCOPN</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>LIBSTACK(0000004096,0000004096,FREE)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>MSGFILE(SYSOUT,FBA,00000121,00000000,NOENQ)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>MSGQ(0000000015)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>NOAUTONIPTSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>OCSSTATUS</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>PAGESIZE(4K,4K,4K)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>NULL</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>PLITASKCOUNT(0000000020)</td>
</tr>
<tr>
<td>PROGRAMMER DEFAULT OVR</td>
<td>POSIX(ON)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>PROFILE(OFF,&quot;&quot;)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>PRUNIT(000000006)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>PURUNIT(000000007)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>RDRUNIT(000000008)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>RECPAD(OFF)</td>
</tr>
<tr>
<td>DD:CEEOPTS OVR</td>
<td>RPTEOPTS(ON)</td>
</tr>
<tr>
<td>DD:CEEOPTS OVR</td>
<td>RPSTG(ON)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>MORTEREUS</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>NOSJMVMD</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>STACK(0000131072,0000131072,ANY,KEEP,0000524288,0000131072)</td>
</tr>
<tr>
<td>DD:CEEOPTS OVR</td>
<td>STORAGE(AA,AA,CC,0000000000)</td>
</tr>
<tr>
<td>DD:CEEOPTS OVR</td>
<td>TERMTHDACT(UAUMP,CESI,000000006)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>NOTEST(ALL,*,PROMPT,INSPPREF)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>THREADHEAP(000000004096,0000004096,FREE)</td>
</tr>
<tr>
<td>INSTALLATION DEFAULT OVR</td>
<td>THREADSTACK(OFF,0000004096,000000004096,FREE)</td>
</tr>
</tbody>
</table>

Figure 13. 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(216168BB)'

ENSM: 21616CD8
+000000 EYE_CATCHER:ENSM ST_HEAP_ALLOC_FLAG:00000001
+000008 ST_HEAP_ALLOC_VAL:AA000000 ST_HEAP_FREE_FLAG:00000000
+000010 ST_HEAP_FREE_VAL:BB000000 REPORT_STORAGE:21615DB4
+000018 UHEAP:CB073C12 21D91018 21D91018 00000000 00000000 21D91010 00010000 00000000 00000000 00 00
+000048 UHEAP:CB073C12 21D90000 22177000 00000000 00000000 00000000 00000000 00000000 00000000 00 00
+000078 UHEAP:CB073C12 21615500 21615500 00020000 00010000 00002000 00001000 00000000 00 00
+0000A8 ENSM_ADDL_HEAPS:21D91018

STSB: 216150B4
+000000 EYE_CATCHER:STSB CRHP_REQ:00000002 DSHP_REQ:00000001
+00000C 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: 21615E4C
+000000 MAX_ALLOC:00009E90 CURR_ALLOC:00003EA0 CURR_ALLOC:00003EA0
+000008 LARGEST:00009E90 GETMAINS:00000000...
+000010 FREEMAINS:00000000

SKSB: 21615E74
+000000 MAX_ALLOC:00009E90 CURR_ALLOC:00003EA0 CURR_ALLOC:00003EA0
+000008 LARGEST:00009E90 GETMAINS:00000000...
+000010 FREEMAINS:00000000

SKSB: 21615E60
+000000 MAX_ALLOC:00009E90 CURR_ALLOC:00003EA0 CURR_ALLOC:00003EA0
+000008 LARGEST:00009E90 GETMAINS:00000000...
+000010 FREEMAINS:00000000

User Heap Control Blocks

HPCB: 21615CF0
+000000 EYE_CATCHER:HPCB FIRST:21D91018 LAST:21D91018

HPDB: 21615D04
+000000 BYTES_ALLOC:00005248 CURR_ALLOC:00005248
+000008 GET_REQ:00000000 FREE_REQ:00000000
+000010 FREEMAINS:00000000

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

HANC: 21D91018
+000000 EYE_CATCHER:HANC NEXT:21615CF0 PREV:21615CF0
+00000C HEAPID:00000000 SEG.Addr:21D91018 ROOT_ADDR:21D96260
+000018 SEG_LEN:00008000 ROOT_LEN:00002000

This is the last heap segment in the current heap.

Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 5 of 18)
Free Storage Tree for Heap Segment 21D91018

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Length</th>
<th>Node</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21D96260</td>
<td>000002DB8</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 21D91018

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

21D91038: Allocated storage element, length=00001030. To display: IP LIST 21D91038 LEN(X'00001030') ASID(X'01A9')

21D92068: Allocated storage element, length=00008028. To display: IP LIST 21D92068 LEN(X'00008028') ASID(X'01A9')

21D92070: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |................................|

21D92890: Allocated storage element, length=0000C000. To display: IP LIST 21D92890 LEN(X'0000C000') ASID(X'01A9')

21D93568: Allocated storage element, length=00002030. To display: IP LIST 21D93568 LEN(X'00002030') ASID(X'01A9')

21D95590: Allocated storage element, length=0000C000. To display: IP LIST 21D95590 LEN(X'0000C000') ASID(X'01A9')

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
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 13. Example of formatted output from LEDATA VERBEXIT (Part 6 of 18)
Map of Heap Segment 21D6D000

To display entire segment: IP LIST 21D6D000 LEN(X'00004000') ASID(X'01A9')

21D6D020: Allocated storage element, length=00000018. To display: IP LIST 21D6D020 LEN(X'00000018') ASID(X'01A9')

Below Heap Control Blocks

HPCB: 21615D50
+000000 EYE_CATCHER:HPCB FIRST:21615D50 LAST:21615D50
   ** NO SEGMENTS ALLOCATED **

HPSB: 21615E1C
+000000 BYTES_ALLOC:00000000 CURR_ALLOC:00000000
+000008 GET_REQ:00000000 FREE_REQ:00000000
+000010 GETMAINS:00000000 FREEMAINS:00000000

Additional Heap Control Blocks

HPSB: 21615E34
+000000 BYTES_ALLOC:00000908 CURR_ALLOC:00000908
+000008 GET_REQ:00000007 FREE_REQ:00000000
+000010 GETMAINS:00000003 FREEMAINS:00000002

ADHP: 22118238
+000000 EYE_CATCHER:ADHP NEXT:F0F00000 HEAPID:22118244

HPCB: 22118244
+000000 EYE_CATCHER:HPCB FIRST:2212B000 LAST:2212B000

HANC: 2212B000
+000000 EYE_CATCHER:HANC NEXT:22118244 PREV:22118244
+00000C HEAPID:22118244 SEG_ADDR:2212B000 ROOT_ADDR:2212B2C8
+000018 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>00000D38</td>
<td></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 2212B000 LEN(X'00001000') ASID(X'01A9')

2212B020: Allocated storage element, length=000002A8. To display: IP LIST 2212B020 LEN(X'000002A8') ASID(X'01A9')

2212B028: D7C3C9C2 00000000 00000000 000102A0 00000000 00000000 00000000 |PCIB............................|

2212B2C8: Free storage element, length=00000D38. To display: IP LIST 2212B2C8 LEN(X'00000D38') ASID(X'01A9')

Summary of analysis for Heap Segment 2212B000:

Amounts of identified storage: Free:00000038 Allocated:000002A8 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.

Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 7 of 18)
Heap Pool Report

QPCB: 21D6DA00
+000080 EYECATCH:QPCB LENGTH:00000800 NUMPOOLS:00000006
+000080C LARGEST_CELL_SIZE:00000000 BIG_REQUESTS:00000001
+000014 STORAGE_HINTS_ADDR:21F00028 FLAGS:E000 NUMGETARRAYS:00
+00001B NUMCELLSIZE:06 GET_POOLINFO_ARRAYS_PTR:21D6DA28

Data for pool 1:
POOLDATA: 21D6DB00
+000000 POOL_INDEX:00000001 INPUT_CELL_SIZE:00000008
+000008 CELL_SIZE:00000010 INPUT_PERCENT:0000000A
+000010 CELL_POOL_SIZE:000000CC CELL_POOL_NUM:000000CC
+000018 POOL_LATCH_ADDR:21EA4BD4 POOL_EXTENTS:00000001
+000020 LAST_CELL:21D962D8 NEXT_CELL:21D9555C
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:00000000
+000030 POOL_NUM_GET_TOTAL:00000002 POOL_NUM_FREE:00000000
+000038 POOL_EXTENTS_ANCHOR:21D95598 POOL_INDEXSAME_SIZE:01
+00003B POOL_INDEX_SIZE:01 POOL_NUMSAME_SIZE:01
+000040 POOL_TRACE_TABLE:21EAF050

Heap Pool Extent Mapping
EXTENT: 21D95598
+000000 EYE_CATCHER:EX31 NEXT EXTENT:00000000

To display entire pool extent: IP LIST 21D95598 LEN(X'00000CC8') ASID(X'01A9')

21D955A0: Allocated storage cell. To display: IP LIST 21D955A0 LEN(X'0000010') ASID(X'01A9')

21D955B8: 21D92C60 AAAAAAAA 00000000 00000000 00000000 00000000 00000000 00000000 |R.-............................|

21D955B8: Allocated storage cell. To display: IP LIST 21D955B0 LEN(X'00000CC8') ASID(X'01A9')

21D955B8: 21D92C60 AAAAAAAA 00000000 00000000 00000000 00000000 00000000 00000000 |R.-............................|

Summary of analysis for Pool 1:
Number of cells: Unused: 202 Free: 56239789 Allocated: 2 Total Used: 204

DE7A703A free cells were not accounted for.

No errors were found while processing this Pool.

Data for pool 2:
POOLDATA: 21D6DC00
+000000 POOL_INDEX:00000002 INPUT_CELL_SIZE:00000020
+000008 CELL_SIZE:00000028 INPUT_PERCENT:0000000A
+000010 CELL_POOL_SIZE:000000CA CELL_POOL_NUM:00000051
+000018 POOL_LATCH_ADDR:21EA4BEC POOL_EXTENTS:00000000
+000020 LAST_CELL:00000000 NEXT_CELL:00000000
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:00000000
+000030 POOL_NUM_GET_TOTAL:00000000 POOL_NUM_FREE:00000000
+000038 POOL_EXTENTS_ANCHOR:00000000 POOL_INDEXSAME_SIZE:01
+00003B POOL_INDEX_SIZE:01 POOL_NUMSAME_SIZE:01
+000040 POOL_TRACE_TABLE:21EAF070

There are no extents for this pool.

Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 8 of 18)
### Data for pool 3:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOLDATA</td>
<td>21D60000</td>
</tr>
<tr>
<td>POOL_INDEX</td>
<td>00000003</td>
</tr>
<tr>
<td>INPUT_CELL_SIZE</td>
<td>00000080</td>
</tr>
<tr>
<td>CELL_SIZE</td>
<td>000000088</td>
</tr>
<tr>
<td>INPUT_PERCENT</td>
<td>0000000A</td>
</tr>
<tr>
<td>CELL_POOL_SIZE</td>
<td>000000CC0</td>
</tr>
<tr>
<td>CELL_POOL_NUM</td>
<td>000000018</td>
</tr>
<tr>
<td>POOL_LATCH_ADDR</td>
<td>21EA48BC</td>
</tr>
<tr>
<td>POOL_INDEX_SAME_SIZE</td>
<td>01</td>
</tr>
<tr>
<td>POOL_INDEX_SIZE</td>
<td>03</td>
</tr>
<tr>
<td>POOL_NUM_SAME_SIZE</td>
<td>01</td>
</tr>
<tr>
<td>POOL_TRACE_TABLE</td>
<td>21F0F090</td>
</tr>
</tbody>
</table>

#### Heap Pool Extent Mapping

EXTENT: 21D92898

To display entire pool extent: IP LIST 21D92898 LEN(X'00000CC8') ASID(X'01A9')

21D928A0: Allocated storage cell. To display: IP LIST 21D928A0 LEN(X'0000088') ASID(X'01A9')

21D928A8: C3C4D3D3 00000000 40000000 00000000 21D92950 00000000 00000000 00000000|CDLL.... ........-...-.8.O.....q|

21D92928: Allocated storage cell. To display: IP LIST 21D92928 LEN(X'0000088') ASID(X'01A9')

21D92930: 00000000 21D92950 00000000 00000000 00000000|.....O...R...|

21D92980: Allocated storage cell. To display: IP LIST 21D92980 LEN(X'0000088') ASID(X'01A9')

21D92988: 00000000 21D92A50 00000000 00000000 21D92C30 00000000 00000000|.....P...R...|

21D92A38: Allocated storage cell. To display: IP LIST 21D92A38 LEN(X'0000088') ASID(X'01A9')

21D92A40: 00000000 21D92A50 00000000 00000000 21D92C30 00000000 00000000|.....P...R...|

21D92AC0: Allocated storage cell. To display: IP LIST 21D92AC0 LEN(X'0000088') ASID(X'01A9')

21D92AC8: 00000000 21D92C30 00000000 00000000 21D92C8C 00000000 00000000|.....R...R...|

21D92B48: Allocated storage cell. To display: IP LIST 21D92B48 LEN(X'0000088') ASID(X'01A9')

21D92B50: 00000000 00000000 00000000 00000000 00000000|................................|

21D92B68: Allocated storage cell. To display: IP LIST 21D92B68 LEN(X'0000088') ASID(X'01A9')

21D92B70: 21D92930 21D955A8 70004000 00000000 21D92C30 00000000 00000000|...R...R...|

Summary of analysis for Pool 3:

- Number of cells: Unused: 16 Free: 56239789 Allocated: 8 Total Used: 24
- DE7A7D3A free cells were not accounted for.
- No errors were found while processing this Pool.

### Data for pool 4:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOLDATA</td>
<td>21D60000</td>
</tr>
<tr>
<td>POOL_INDEX</td>
<td>00000004</td>
</tr>
<tr>
<td>INPUT_CELL_SIZE</td>
<td>00000100</td>
</tr>
<tr>
<td>CELL_SIZE</td>
<td>00000108</td>
</tr>
<tr>
<td>INPUT_PERCENT</td>
<td>0000000A</td>
</tr>
<tr>
<td>CELL_POOL_SIZE</td>
<td>000000CC0</td>
</tr>
<tr>
<td>CELL_POOL_NUM</td>
<td>0000000C0</td>
</tr>
<tr>
<td>POOL_LATCH_ADDR</td>
<td>21EA4C10</td>
</tr>
<tr>
<td>POOL_INDEX_SAME_SIZE</td>
<td>01</td>
</tr>
<tr>
<td>POOL_INDEX_SIZE</td>
<td>04</td>
</tr>
<tr>
<td>POOL_TRACE_TABLE</td>
<td>21F3F0B0</td>
</tr>
</tbody>
</table>

There are no extents for this pool.

---

Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 9 of 18)
Data for pool 5:
POOLDATA: 21D6DF00
+000000 POOL_INDEX:00000005 INPUT_CELL_SIZE:00000400
+000008 CELL_SIZE:00000408 INPUT_PERCENT:0000000A
+000010 CELL_POOL_SIZE:00000800 CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:21EA4C24 POOL_EXTENSIONS:00000001
+000020 LAST_CELL:21D91C60 NEXT_CELL:21D91858
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:00000000
+000030 POOL_NUM_GET_TOTAL:00000002 POOL_NUM_FREE:00000000
+000038 POOL_EXTENSIONS_ANCHOR:21D91040 POOL_INDEX_SAME_SIZE:01
+00003D POOL_INDEX_SIZE:05 POOL_NUM_SAME_SIZE:01
+000040 POOL_TRACE_TABLE:21F6F0D0

Heap Pool Extent Mapping
EXTENT: 21D91040
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000

To display entire pool extent: IP LIST 21D91040 LEN(X'00001028') ASID(X'01A9')

21D91048: Allocated storage cell. To display: IP LIST 21D91048 LEN(X'00000408') ASID(X'01A9')

21D91050: 21D92070 21D92258 21D92295 21D92202 21D9230F 21D9234C 21D92389 21D923C6 ....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R....R...
Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 11 of 18)
To display entire DSA: IP LIST 21D71020 LEN(X'0000030F8') ASID(X'01A9')

DSA: 21D71248
+000000  FLAGS:10CC MEMD:CCCC BKC:21D71130 FWC:21D71328
+000000C R14:A1608A66 R15:21D7145C R0:216011A4
+0000018 R1:21D71DE0 R2:21D713F0 R3:21600DFA
+0000024 R4:21D71310 R5:21D71310 R6:21D71310
+0000030 R7:21D71310 R8:00000003 R9:00000000
+0000048 LWS:00000000 NAB:21D71328 PNAB:CCCCCCCC
+0000064 RENT:CCCCCCCC CILC:CCCCCCCC MODE:CCCCCCCC
+0000078 MM:CCCCCCCC

Contents of DSA at location 21D71248:
+00000000 10CCCCCC 21D71130 21D71328 A1608A66 21D7145C 216011A4 21D71DE0 21D713F0 |.....P...-....u*.-.u.P...P..|
+00000020 21600DFA 21D71310 21D71310 21D71310 21D71310 21D71310 21D71310 21D71310 |.-...P...-...P...P...........KY.|
+00000040 A1692F48 21616BB8 00000000 21D71328 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |...../,......P..................|
+00000060 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+00000080 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000A0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |.........................-.u.-.n|
+000000C0 00000003 21600EDC 22128948 00000000 220B1178 220B23B8 00000000 00000000 |.....-....i.....................|
+000000E0 00000000 00000002 22128E40 00000000 00000000 00000000 21D6E148 CCCCCCCC |........... .............O......|

DSA: 21D71130
+000000  FLAGS:10CC MEMD:CCCC BKC:21D71030 FWC:CCCCCCCC
+000000C R14:A1608A66 R15:21D7145C R0:216011A4
+0000018 R1:21D71DE0 R2:21D71310 R3:21D71310
+0000024 R4:21D71310 R5:21D71310 R6:21D71310
+0000030 R7:21D71310 R8:00000003 R9:00000000
+0000048 LWS:00000000 NAB:21D71130 PNAB:CCCCCCCC
+0000064 RENT:CCCCCCCC CILC:CCCCCCCC MODE:CCCCCCCC
+0000078 MM:CCCCCCCC

Contents of DSA at location 21D71130:
+00000000 10CCCCCC 21D71030 CCCCCCCC A1608A66 21D7145C 216011A4 21D71DE0 21D71310 |.....P...-.y.-..u*.-.u.P...P..|
+00000020 00000002 A1692F48 216157D0 216039EC 21604330 00000030 00000008 A1D2E8B2 |........./...-...-...........KY.|
+00000040 A1692F48 21616BB8 00000000 21D71130 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |...../,......P..................|
+00000060 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+00000080 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000A0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000C0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000E0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|

To display entire DSA: IP LIST 21D71020 LEN(X'000000118') ASID(X'01A9')

DSA: 21D71030
+000000  FLAGS:0000 MEMD:CCCC BKC:21617660 FWC:CCCCCCCC
+000000C R14:A169310E R15:21D71E8BE R0:7D000009
+0000018 R1:21D71E18 R2:21D71E8BE R3:21D71E8BE
+0000024 R4:A169303A R5:216157D0 R6:216039EC
+0000030 R7:21604330 R8:00000003 R9:00000008
+0000048 LWS:00000000 NAB:21D71130 PNAB:CCCCCCCC
+0000064 RENT:CCCCCCCC CILC:CCCCCCCC MODE:CCCCCCCC
+0000078 MM:CCCCCCCC

Contents of DSA at location 21D71030:
+00000000 0000CCCC 21617660 CCCCCCCC A169310E 21D71E8BE 7D000009 21D71DE0 21D71310 |...../.-.........KY.'....P...-..|
+00000020 00000002 A169303A 216157D0 216039EC 21604330 00000030 00000008 A1D2E8B2 |........./...-...-...........KY.|
+00000040 A1692F48 21616BB8 00000000 21D71130 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |...../,......P..................|
+00000060 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+00000080 21D6E1E8 00000000 00000000 CCCCCCCC 00000001 CCCCCCCC CCCCCCCC CCCCCCCC |.O.Y............................|
+000000A0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000C0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000E0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|

User Stack Control Blocks
STKH: 21D71018
+000000  EYE_CATCHER:STKU NEXT:2161742C PREV:2161742C
+00000C SEGMENT_LEN:00020000

Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 12 of 18)
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STKH: 00015000</td>
<td>Library Stack Control Blocks</td>
</tr>
<tr>
<td>+000000 EYE_CATCHER:STKL</td>
<td>NEXT:21617470 PREV:21617470</td>
</tr>
<tr>
<td>+00000C SEGMENT_LEN:00001000</td>
<td></td>
</tr>
<tr>
<td>STKL:</td>
<td></td>
</tr>
<tr>
<td>+000000 EYE_CATCHER:STKL</td>
<td>NEXT:21617470 PREV:21617470</td>
</tr>
<tr>
<td>+00000C SEGMENT_LEN:00001000</td>
<td></td>
</tr>
<tr>
<td>HCOM: 2160CF38</td>
<td>Condition Management Control Blocks</td>
</tr>
<tr>
<td>+000000 PICA_AREA:00000000 00000000</td>
<td>EYES:HCOM CAA_PTR1:21616BB8</td>
</tr>
<tr>
<td>+000014 CVTDCB:9B</td>
<td>FLAG:60F0C000 EXIT_STK:22126E68</td>
</tr>
<tr>
<td>+000020 RSM_PTR:00000000</td>
<td>HDLL_STK:22126E68</td>
</tr>
<tr>
<td>+000028 SRP_TOKEN:00000000</td>
<td>HCHK5_RESULTS:00001100 SHUNT_VALIDFLAG:00000001</td>
</tr>
<tr>
<td>+00009C SHUNT_PSW:078D2000 A16FB7CA</td>
<td>SHUNT_REG0:00000001 SHUNT_REG1:21D77050</td>
</tr>
<tr>
<td>+000480 SHUNT_CODE1:00000000</td>
<td>SHUNT_CODE2:00000004</td>
</tr>
<tr>
<td>CIBH: 21D72B28</td>
<td>Machine State</td>
</tr>
<tr>
<td>+000000 EYE:CIBH BACK:2160E430</td>
<td>FRWD:00000000</td>
</tr>
<tr>
<td>+000010 CIB:00000000 00000000</td>
<td>STATE:00000000 PRM_DESC:00000000</td>
</tr>
<tr>
<td>+000084 COND_DEFAULT:00000000</td>
<td>Q_DATA_TOKEN:00000000 FDBK:00000000</td>
</tr>
<tr>
<td>+000220 BBRANCH_STMTID:........</td>
<td>BBRANCH_STMTLEN:0000</td>
</tr>
<tr>
<td>+000248 MCH_EYE:....</td>
<td></td>
</tr>
<tr>
<td>+000260 GPR00:00000000</td>
<td>GPR01:00000000</td>
</tr>
<tr>
<td>+000280 GPR02:00000000</td>
<td>GPR03:00000000</td>
</tr>
<tr>
<td>+0002A0 FLT:0:00000000 00000000</td>
<td>FLT:0:00000000 00000000</td>
</tr>
<tr>
<td>+0002A8 FLT:2:00000000 00000000</td>
<td>FLT:2:00000000 00000000</td>
</tr>
<tr>
<td>+0002B0 FLT:4:00000000 00000000</td>
<td>FLT:4:00000000 00000000</td>
</tr>
<tr>
<td>+0002BC FLT:6:00000000 00000000</td>
<td>FLT:6:00000000 00000000</td>
</tr>
<tr>
<td>+000300 FLT:0:00000000 00000000</td>
<td>FLT:0:00000000 00000000</td>
</tr>
<tr>
<td>+000308 FLT:2:00000000 00000000</td>
<td>FLT:2:00000000 00000000</td>
</tr>
<tr>
<td>+000310 FLT:4:00000000 00000000</td>
<td>FLT:4:00000000 00000000</td>
</tr>
<tr>
<td>+000318 FLT:6:00000000 00000000</td>
<td>FLT:6:00000000 00000000</td>
</tr>
<tr>
<td>+000320 FLT:8:00000000 00000000</td>
<td>FLT:8:00000000 00000000</td>
</tr>
<tr>
<td>+000328 FLT:10:00000000 00000000</td>
<td>FLT:10:00000000 00000000</td>
</tr>
<tr>
<td>+000330 FLT:12:00000000 00000000</td>
<td>FLT:12:00000000 00000000</td>
</tr>
<tr>
<td>+000338 FLT:14:00000000 00000000</td>
<td>FLT:14:00000000 00000000</td>
</tr>
</tbody>
</table>

Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 13 of 18)
Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 14 of 18)
Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 15 of 18)
Message Processing Control Blocks

CMXB: 216151A0
+000000 EYE:CMXB SIZE:0148  FLAGS:0000  DHEAD1:00016000
+00000C DHEAD2:00012000

MDST forward chain from CMXBDHEAD(1)

MDST: 00016000
+000000 EYE:MDST SIZE:0100  CTL:40  CEEUMPLC:00
+00000C NEXT:00000000  PREV:00016000  DONAM:CEEDUMP

MDST: 00012000
+000000 EYE:MDST SIZE:0100  CTL:40  CEEUMPLC:00
+00000C NEXT:00000000  PREV:00016000  DONAM:SYSOUT

MDST back chain from CMXBDHEAD(2)

MDST: 00012000
+000000 EYE:MDST SIZE:0100  CTL:40  CEEUMPLC:00
+00000C NEXT:00000000  PREV:00016000  DONAM:SYSOUT

MDST: 00016000
+000000 EYE:MDST SIZE:0100  CTL:40  CEEUMPLC:00
+00000C NEXT:00000000  PREV:00016000  DONAM:CEEDUMP

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

MGF: 22167028
+000000 EYE:MGF PREV:22131780  NEXT:22118380

MGF: 22118380
+000000 EYE:MGF PREV:22167028  NEXT:2160F080

MGF: 2160F080
+000000 EYE:MGF PREV:22118380  NEXT:221315C0

MGF: 221315C0
+000000 EYE:MGF PREV:2160F080  NEXT:22131780

MGF: 22131780
+000000 EYE:MGF PREV:221315C0  NEXT:22167028

Information for enclave main

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

Registers and PSW:

GPR0..... 00000000_84000000  GPR1..... 00000000_84000000  GPR2..... 00000000_21D72618  GPR3..... 00000000_00020009
GPR4..... 00000000_216C9A88  GPR5..... 00000000_216D9C32  GPR6..... 00000000_21655220  GPR7..... 00000000_2160E430
GPR8..... 00000000_21D72618  GPR9..... 00000000_21D721AC  GPR10..... 00000000_21D7201F  GPR11..... 00000000_A169A858
GPR12..... 00000000_21D61E88  GPR13..... 00000000_21D4E11E  GPR14..... 00000000_A160980E  GPR15..... 00000000_00000000
PSW..... 078D1400 A1609C32

Figure 13. Example of formatted output from LEDATA VERBEXIT (Part 16 of 18)
Control Blocks Associated with the Thread:

Figure 13. Example of formatted output from LEDATA 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 Language Environment 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>
## Table 21. Contents of the Language Environment LEDATA VERBEXIT formatted output (continued)

<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) |
| **[3]** CEECAA            | Formats the contents of the Language Environment common anchor area (CAA). See “Common Anchor Area” on page 63 for a description of the fields in the CAA. |
| **[4]** CEEDLLF           | Formats the contents of all Language Environment CEEDLLF (DLLF) control blocks that are in use. See The 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. |
| **[6]** CEERCB            | Formats the contents of the Language Environment region control block (RCB). |
| **[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]** Run-Time Options  | Lists the run-time 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 112. |
| **[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 63 or “Downward-growing (XPLINK) stack frame section” on page 64 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 74 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. |
| **[14]-[17]** NTHREADS information: 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 50. |
| **[14] - [21]** CEEDUMP Formatted Control Blocks: These sections are included when the CEEDUMP parameter is specified on the LEDATA invocation. |
Table 21. Contents of the Language Environment LEDATA VERBEXIT formatted output (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<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 CEEPAA macro.  
  • Service level: The latest service level applied to the compile unit (for example, for IBM products, it would be the PTF number).  
  • Status: Routine status can be call, exception, or running.  

The second 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.  
  • 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: Contains the year, month and day in which the routine was compiled.  
  • Attributes: The available compilation attributes of the compile unit include:  
    – A label identifying the LE-supported language such as COBOL, ENT PL/I, C/C++, and so on.  
    – 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.  
    – POSIX, If the CEEDUMP was created under a POSIX environment. |
<p>| [18] Control Blocks Associated with the Thread | Lists the contents of the thread synchronization queue element (SQEL). |</p>
<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[19] Enclave Control Blocks</td>
<td>If the POSIX run-time 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 run-time 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>[20] Language Environment Trace Table</td>
<td>If the TRACE run-time option was set to ON, this section shows the contents of the Language Environment trace table.</td>
</tr>
<tr>
<td>[21] Process Control Blocks</td>
<td>If the POSIX run-time option was set to ON, this section lists the contents of the process level latch table.</td>
</tr>
<tr>
<td>[22] Preinitialization Information</td>
<td>This section is included when the PTBL parameter is specified on the LEDATA invocation. This section formats information related to preinitialization. See <a href="#">PTBL LEDATA output</a> for more information. If the preinitialization service CEEPIPI was not used to initialize this environment, the message: No PIPICB associated with CAA is displayed instead.</td>
</tr>
</tbody>
</table>

**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. See PTBL LEDATA output for more information. If the preinitialization service CEEPIPI was not used to initialize this environment, the message: No PIPICB associated with CAA is displayed instead.
Language Environment Product 04 V01 R09.00

PreInitialization Programming Interface Trace Data
CEEPPI Environment Table Entry and Trace Entry:
  Active CEEPIPI Environment ( Address 20905C08 )
  Eyecatcher : CEEXIPTB
  TCB address : 00BBDE08

CEEPPI Environment:
  Non-XPLINK Environment
  Environment Type : MAIN
  Sequence of Calls not active
  Exits not established
  Signal Interrupt Routines not registered
  Service Routines are 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 ( 20905C08 )

    +000000 20905C08 A0910620 20919B30 80000000 00000000 00000000 00000000 00000000 00000000 |.j...j..........................|
    +000020 20905DD8 00000000 00000000 00000000 A0910530 00000003 209197D8 00000003 20910530 |.............j.......jpQ.....j..|
    +000040 20905DF8 A0910530 00009AD0 C9E2D1D7 D7C3C1F3 00000000 00000000 00000000 00000000 |.j......ISJPPCA3................|

  CEEPIPI Table Index 1 ( Entry 2 ) not in use.
  CEEPIPI Table Index 2 ( Entry 3 ) not in use.

Figure 14. 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 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. Figure 15 on page 113 illustrates the output produced by specifying the HEAP option. "Heap report sections of the LEDATA output" on page 116 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 116. 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.

Figure 14. Example of formatted PTBL output from LEDATA VERBEXIT (Part 2 of 2)
Heap Storage Control Blocks

ENSM: 00014D30
+0000A8 ENSM_ADDL_HEAPS:259B1120

User Heap Control Blocks

HPCB: 00014D4B
+000000 EYE_CATCHER:HPCB FIRST:25995000 LAST:25995000

HANC: 25995000
+000000 EYE_CATCHER:HANC NEXT:00014D4B PREV:00014D4B
+000000C HEAPID:00000000 SEG_ADDR:25995000 ROOT_ADDR:25995080
+000018 SEG_LEN:00000000 ROOT_LEN:00007F50

This is the last heap segment in the current heap.

Free Storage Tree for Heap Segment 25995000

<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>259950B0</td>
<td>00007F50</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 25995000

To display entire segment: IP LIST 25995000 LEN(X'00008000') ASID(X'0021')
25995020: Allocated storage element, length=00000038. To display: IP LIST 25995020 LEN(X'00000038') ASID(X'0021')
25995028: 1C34D3D3 00000000 40000000 00000000 24700F90 24703F70 25993870 00008490 |CDLL.. ..........q.....r......|
25995058: Allocated storage element, length=00000038. To display: IP LIST 25995058 LEN(X'00000038') ASID(X'0021')
25995068: 1C34D3D3 25995028 80000000 00000000 247006F0 24700770 2471CEB0 00000150 |CDLL.r&............0...........&|
25995090: Allocated storage element, length=00000010. To display: IP LIST 25995090 LEN(X'00000010') ASID(X'0021')
25995098: 259ADBB8 00000000 |........ |
259950A0: Allocated storage element, length=00000010. To display: IP LIST 259950A0 LEN(X'00000010') ASID(X'0021')
259950A8: 259ADEE0 00000000 |........ |
259950B0: Free storage element, length=00007F50. To display: IP LIST 259950B0 LEN(X'00007F50') 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.

Free Storage Tree for Heap Segment 24A91000

The free storage tree is empty.

Figure 15. Example formatted detailed heap segment report from LEDATA VERBEXIT (Part 1 of 4)
Map of Heap Segment 24A91000

To display entire segment: IP LIST 24A91000 LEN(X'00F00008') ASID(X'0021')

24A91020: Allocated storage element, length=00F00008. To display: IP LIST 24A91020 LEN(X'00F00008') ASID(X'0021')

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.

Summary of analysis for Heap Segment 259AC000:
Amounts of identified storage: Free:00000FE8 Allocated:00000FF8 Total:00001FE0
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.

Figure 15. Example formatted detailed heap segment report from LEDATA VERBEXIT (Part 2 of 4)
Below Heap Control Blocks

HPCB: 0004DA8
+000000 EYE_CATCHER:HPCB FIRST:00044000 LAST:00044000

HANC: 0004DA0
+000000 EYE_CATCHER:HANC NEXT:00014DA8 PREV:00014DA8
+00000C HEAPID:00014DA8 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>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>00044388</td>
<td>0001C78</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(X'00002000') ASID(X'0021')

00044020: Allocated storage element, length=00000048. To display: IP LIST 00044020 LEN(X'00000048') ASID(X'0021')
00044028: CBC4DE3E 00000000 00044220 00000040 00010000 00000001 000241E0 24701038 | HDLS........... ................|

00044068: Allocated storage element, length=00000128. To display: IP LIST 00044068 LEN(X'00000128') ASID(X'0021')
00044078: 07000700 05E0990F E0A641DE 02258C0 E125B80F E116080F E2067D0 01200001 | ........M...........0....SFXM....|

00044190: Allocated storage element, length=00000088. To display: IP LIST 00044190 LEN(X'00000088') ASID(X'0021')
00044198: C3E2E3D2 00000000 00000000 00000001 00000001 00000068 04000000 00000000 | CSTK............................|

00044218: Allocated storage element, length=00000048. To display: IP LIST 00044218 LEN(X'00000048') ASID(X'0021')
00044228: CBC4DE3E 00044028 00000000 00000040 00010000 00000002 000241E0 259A0890 | HDLS... ........ ................|

00044268: Allocated storage element, length=00000128. To display: IP LIST 00044268 LEN(X'00000128') ASID(X'0021')
00044268: 07000700 05E0990F E0A641DE 02258C0 E125B80F E116080F E2067D0 01200001 | ........M...........0....SFXM....|

00044388: Free storage element, length=00001C78. To display: IP LIST 00044388 LEN(X'00001C78') ASID(X'0021')

Summary of analysis for Heap Segment 00044000:
Amounts of identified storage: Free:00001C78 Allocated:00000368 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:259ADCO8 HEAPID:259B24B4

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

ADHP: 259ADCO8
+000000 EYE_CATCHER:ADHP NEXT:FD000000 HEAPID:259ADCE14

HPCB: 259ADCE14
+000000 EYE_CATCHER:hpcb FIRST:259AE000 LAST:259AE000

HANC: 259AE000
+000000 EYE_CATCHER:HANC NEXT:259ADCE14 PREV:259ADCE14
+00000C HEAPID:259ADCE14 SEG_ADDR:259AE000 ROOT_ADDR:259AEE1E8
+000018 SEG_LEN:00001000 ROOT_LEN:00008E18

Figure 15. Example formatted detailed heap segment report from LEDATA VERBEXIT (Part 3 of 4)
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 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 |
Table 22. Contents of the Heap 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 Segment Map Report</td>
<td>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 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:</td>
</tr>
<tr>
<td></td>
<td>• Is a multiple of 8</td>
</tr>
<tr>
<td></td>
<td>• Is not zero</td>
</tr>
<tr>
<td></td>
<td>• Is not larger than the heap segment length</td>
</tr>
<tr>
<td></td>
<td>• Does not cause the end of the element to fall outside of the current heap segment</td>
</tr>
<tr>
<td></td>
<td>• Does not cause the element to overlap a free storage node</td>
</tr>
<tr>
<td></td>
<td>If the heap_free_value of the STORAGE run-time 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.</td>
</tr>
</tbody>
</table>

**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 run-time 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 run-time 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 run-time 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 HEAPPOOL LEDATA output**

The Language Environment IPCS VERBEXIT LEDATA generates a detailed HEAPPOOL report when HEAPPOOL is ON. The detailed HEAPPOOL report is useful when trying to find potential damaged cells because it provides very specific information. Figure 16 on page 119 shows an example of a report and “HEAPPOOLS report sections of the LEDATA output” on page 123 describes the information contained in the formatted output.
Figure 16. Example formatted detailed HEAPPOOL report from LEDATA VERBEXIT (Part 1 of 4)
Figure 16. Example formatted detailed HEAPPOOL report from LEDATA VERBEXIT (Part 2 of 4)
Figure 16. Example formatted detailed HEAPPool report from LEDATA VERBEXIT (Part 3 of 4)
Data for pool 5.4:
POOLDATA: 25C1F500
+000000 POOL_INDEX:00000000 INPUT_CELL_SIZE:00000000
+000008 CELL_SIZE:00000000 INPUT_PERCENT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000018 POOL_LATCH_ADDR:25C54C24 POOL_EXTENTS:00000000
+000020 LAST_CELL:25C48048 NEXT_CELL:25C48048
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:25C47430
+000030 POOL_NUM_GET_TOTAL:00000000 POOL_NUM_FREE:00000000
+000038 POOL_EXTENTS_ANCHOR:25C47428 POOL_INDEX_SAME_SIZE:00
+00003D POOL_INDEX_SIZE:00 POOL_NUM_SAME_SIZE:00
+000040 POOL_TRACE_TABLE:25DA6140

EXTENT: 25C47428
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire pool extent: IP LIST 25C47428 LEN(X'00001028') ASID(X'0020')
25C47430: Free storage cell. To display: IP LIST 25C47430 LEN(X'00000408') ASID(X'0020')
25C47438: Free storage cell. To display: IP LIST 25C47438 LEN(X'00000408') ASID(X'0020')
25C47440: Free storage cell. To display: IP LIST 25C47440 LEN(X'00000408') 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
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 5.5:
POOLDATA: 25C1F600
+000000 POOL_INDEX:00000000 INPUT_CELL_SIZE:00000000
+000008 CELL_SIZE:00000000 INPUT_PERCENT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000018 POOL_LATCH_ADDR:25C54C38 POOL_EXTENTS:00000000
+000020 LAST_CELL:25C49078 NEXT_CELL:25C49078
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:25C48460
+000030 POOL_NUM_GET_TOTAL:00000000 POOL_NUM_FREE:00000000
+000038 POOL_EXTENTS_ANCHOR:25C48458 POOL_INDEX_SAME_SIZE:00
+00003D POOL_INDEX_SIZE:00 POOL_NUM_SAME_SIZE:00
+000040 POOL_TRACE_TABLE:25DA6160

EXTENT: 25C48458
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire pool extent: IP LIST 25C48458 LEN(X'00001028') ASID(X'0020')
25C48460: Free storage cell. To display: IP LIST 25C48460 LEN(X'00000408') ASID(X'0020')
25C48468: Free storage cell. To display: IP LIST 25C48468 LEN(X'00000408') ASID(X'0020')
25C48470: Free storage cell. To display: IP LIST 25C48470 LEN(X'00000408') 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
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 6:
POOLDATA: 25C1F700
+000000 POOL_INDEX:00000000 INPUT_CELL_SIZE:00000000
+000008 CELL_SIZE:00000000 INPUT_PERCENT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000018 POOL_LATCH_ADDR:25C54C48 POOL_EXTENTS:00000000
+000020 LAST_CELL:25C452E8 NEXT_CELL:25C452E8
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:25C442D8
+000030 POOL_NUM_GET_TOTAL:00000321 POOL_NUM_FREE:00000000
+000038 POOL_EXTENTS_ANCHOR:25C43AC8 POOL_INDEX_SAME_SIZE:00
+00003D POOL_INDEX_SIZE:00 POOL_NUM_SAME_SIZE:00
+000040 POOL_TRACE_TABLE:25E06180

EXTENT: 25C43AC8
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire pool extent: IP LIST 25C43AC8 LEN(X'00001028') ASID(X'0020')
25C43AD0: Allocated storage cell. To display: IP LIST 25C43AD0 LEN(X'00000008') ASID(X'0020')
25C43B8: Free storage cell. To display: IP LIST 25C43B8 LEN(X'00000008') ASID(X'0020')
25C442D8: Free storage cell. To display: IP LIST 25C442D8 LEN(X'00000008') ASID(X'0020')

[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: 2 Allocated: 1 Total Used: 4
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Figure 16. Example formatted detailed HEAPPOOL report from LEDATA VERBEXIT (Part 4 of 4)
HEAPPOOLS report sections of the LEDATA output
The HEAPPOOLS 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 HEAPPOOLS 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>
<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 HEAPPOOLS trace LEDATA output
The Language Environment IPCS VERBEXIT LEDATA generates a detailed HEAPPOOLS 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.

<table>
<thead>
<tr>
<th>HPT(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANGUAGE ENVIRONMENT DATA</td>
</tr>
<tr>
<td>Language Environment Product 04 V01 R10.00</td>
</tr>
</tbody>
</table>

[1] HEAPPOOLS Trace Table


[4] Type: FREE Cell Address: 25E91AC0 Cpuid: 01 Tcb: 008AFCF0

<table>
<thead>
<tr>
<th>CALL NAME</th>
<th>CALL ADDRESS</th>
<th>CALL OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetStorage::“GetStorage()</td>
<td>25E53360</td>
<td>00000000</td>
</tr>
<tr>
<td>foo8()</td>
<td>25E53598</td>
<td>00000000</td>
</tr>
<tr>
<td>foo7()</td>
<td>25E53678</td>
<td>0000005A</td>
</tr>
<tr>
<td>foo6()</td>
<td>25E536F0</td>
<td>0000005A</td>
</tr>
<tr>
<td>foo5()</td>
<td>25E53768</td>
<td>0000005A</td>
</tr>
<tr>
<td>foo4()</td>
<td>25E537E0</td>
<td>0000005A</td>
</tr>
<tr>
<td>foo3()</td>
<td>25E53858</td>
<td>0000005A</td>
</tr>
<tr>
<td>foo2()</td>
<td>25E53800</td>
<td>0000005A</td>
</tr>
<tr>
<td>foo1()</td>
<td>25E53948</td>
<td>0000005A</td>
</tr>
<tr>
<td>thread</td>
<td>25E53A50</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Figure 17. Example formatted detailed HEAPPOOLS trace report from LEDATA VERBEXIT (Part 1 of 4)
Figure 17. Example formatted detailed HEAPPOOLS trace report from LEDATA VERBEXIT (Part 2 of 4)
Figure 17. Example formatted detailed HEAPPOOLS trace report from LEDATA VERBEXIT (Part 3 of 4)
Table 24. Contents of HEAPPOOLS 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. Note: The trace wraps for each poolid after a specific number of entries. The number of entries is controlled by the HEAPCHK run-time 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 run-time 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 17 on page 127 illustrates the C/C++-specific output produced. Figure 5 on page 43 and Table 25 on page 148 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.
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 1 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 2 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 3 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 4 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 5 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 6 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 7 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 8 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 9 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 10 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 11 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 12 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 13 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 14 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 15 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 16 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 17 of 22)
**Figure 18.** Example formatted C/C++ output from LEDATA VERBEXIT (Part 18 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 19 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 20 of 22)
Figure 18. Example formatted C/C++ output from LEDATA VERBEXIT (Part 21 of 22)
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>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 (FSCE), which represents the specific type of IO being performed. The following is a list of FSCEs that 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 about the DCB, see z/OS DFSMS Macro Instructions for Data Sets.</td>
</tr>
<tr>
<td>DCBE</td>
<td>The data control block extension. For more information about the DCBE, see z/OS DFSMS Macro Instructions for Data Sets.</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>
<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. Figure 19 on page 150 illustrates the COBOL-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) run-time option. Table 26 on page 151 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 19. Example formatted COBOL output from LEDATA VERBEXIT (Part 1 of 2)
Table 26. 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>
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. Figure 20 illustrates the PL/I-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) run-time option. Table 27 on page 158 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 20. Example formatted PL/I output from LEDATA VERBEXIT (Part 1 of 7)
Figure 20. Example formatted PL/I output from LEDATA VERBEXIT (Part 2 of 7)
Figure 20. Example formatted PL/I output from LEDATA VERBEXIT (Part 3 of 7)
Figure 20. Example formatted PL/I output from LEDATA VERBEXIT (Part 4 of 7)
Figure 20. Example formatted PL/I output from LEDATA VERBEXIT (Part 5 of 7)
Figure 20. Example formatted PL/I output from LEDATA VERBEXIT (Part 6 of 7)
Table 27. Contents of PL/I-specific sections of LEDATA output

<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

```plaintext
CBF address STRUCTure (cbname)
```

**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 28 on page 160](#). 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 21 on page 160](#).

```
CBF 213F6B48 struct(CEECAA)
```
For more information on using the IPCS CBF command, refer to the "CBFORMAT subcommand" section in z/OS MVS IPCS Commands, SA22-7594.

Table 28. 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>CEEMX8</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>CEEDSAATR</td>
<td>XPLINK Transition Area</td>
</tr>
<tr>
<td>CEEDSAUS</td>
<td>Dynamic Storage Area (XPLINK style)</td>
</tr>
</tbody>
</table>
Table 28. 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>CEEENSM</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>CEEMDST</td>
<td>Message Destination</td>
</tr>
<tr>
<td>CEEMGF</td>
<td>Mapping of the Message Formatter (IBM1MGF)</td>
</tr>
<tr>
<td>CEEPCB</td>
<td>Process Control Block</td>
</tr>
<tr>
<td>CEEPMB</td>
<td>Program Management Control Block</td>
</tr>
<tr>
<td>CEERCB</td>
<td>Region Control Block</td>
</tr>
<tr>
<td>CEESKSB</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>

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 run-time 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 run-time option is set to ON.
- The TRACE run-time option is set to NODUMP and the TERMTHDACT run-time option is set to DUMP, UADUMP, TRACE, or UATRACE.
- The TRACE run-time option is set to DUMP (the default).

For more information about the CEE3DMP callable service, the TERMTHDACT run-time option, or the TRACE run-time option, see [z/OS Language Environment Programming Reference](#).

The TRACE run-time option activates Language Environment run-time 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 run-time option. Table 29 summarizes the dump contents that are generated under abnormal termination.

**Table 29. TERMTHDACT run-time option settings and dump contents produced**

<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>
<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 run-time 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 run-time 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:

<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 22. Format of the trace table entry*
**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>05</td>
<td>COBOL</td>
</tr>
<tr>
<td>07</td>
<td>Fortran</td>
</tr>
<tr>
<td>08</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>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 run-time 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 run-time 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 run-time 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>
<tr>
<td>20</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>
</tbody>
</table>

When LE=1 is specified: Table 30 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 213.

<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>
</tbody>
</table>
When LE=2 is specified: Table 31 shows the Language Environment records that may be generated.

### Table 31. 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>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>
<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>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>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>
<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>EUO</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>
</tbody>
</table>
### Table 31. 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>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>
<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 unitialized</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>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>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>
<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>
</tbody>
</table>
### Table 31. 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>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 32 shows the format for the Mutex – Condition Variable – Latch entries in the trace table.

### Table 32. Format of the mutex/CV/latch records

<table>
<thead>
<tr>
<th>Class</th>
<th>Source</th>
<th>Event</th>
<th>Object Addr</th>
<th>Name1</th>
<th>Name2</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 31 on page 165

**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 33. 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 213.

Table 33. 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 34 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 213.

Table 34. 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

Figure 23 on page 171 shows an example of a dump of the trace table when you specify the LE=1 suboption (the library call/return trace).
Figure 23. Trace table in dump output (LE=1 suboption)
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 run-time 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 run-time options: TRAP(ON, NOSPIE) and TERMTHDACT(UAIMM). Setting these run-time 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 24 shows the structure as it appears in stdio.h.

```c
typedef struct __amrc_type {
  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 __str[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 long __XRBA;
    #else
    unsigned int __XRBA1;
    unsigned int __XRBA2;
    #endif
  unsigned char __amrc_noseek_to_seek;
  char __amrc_pad[23];
  #endif
} __amrc_type;
```
Figure 25 shows the __amrc2 structure as it appears in stdio.h.

```c
struct {
    long int __error2;
    char __pad_error2[4];
    FILE *__fileptr;
    long int __reserved[6];
};
```

1. **__error2**
   - 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 35 on page 178.

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++ Run-Time
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 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().

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 35 lists __last_op values you could receive and where
to look for further information.

Table 35. __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>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------</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>
<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 CAMLIST OBTAIN.</td>
</tr>
<tr>
<td>__IO_LOCATE</td>
<td>Sets __error with return code from I/O CAMLIST LOCATE.</td>
</tr>
<tr>
<td>__IO/catalog</td>
<td>Sets __error with return code from I/O CAMLIST CAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>__SVC99_ALLOC</td>
<td>Sets __alloc structure with info and error codes from SVC 99 allocation.</td>
</tr>
<tr>
<td>__SVC99_ALLOC_NEW</td>
<td>Sets __alloc structure with info and error codes from SVC 99 allocation of NEW file.</td>
</tr>
<tr>
<td>__SVC99_UNALLOC</td>
<td>Sets __unalloc structure with info and error codes from SVC 99 unallocation.</td>
</tr>
<tr>
<td>__C_TRUNCATE</td>
<td>Set when C or C++ truncates output data. Usually, this is data written to a text file with no newline such that the record fills up to capacity and subsequent characters cannot be written. For a record I/O file, this refers to an fwrite() writing more data than the record can hold. Truncation is always rightmost data. There is no return code.</td>
</tr>
<tr>
<td>__C_FCBCHECK</td>
<td>Set when C or C++ FCB is corrupted. This is due to a pointer corruption somewhere. File cannot be used after this.</td>
</tr>
<tr>
<td>__C_DBCS_TRUNCATE</td>
<td>This occurs when writing DBCS data to a text file and there is no room left in a physical record for anymore double byte characters. A new-line is not acceptable at this point. Truncation will continue to occur until an SI is written or the file position is moved. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_SO_TRUNCATE</td>
<td>This occurs when there is not enough room in a record to start any DBCS string or else when a redundant SO is written to the file before an SI. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_SI_TRUNCATE</td>
<td>This occurs only when there was not enough room to start a DBCS string and data was written anyways, with an SI to end it. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
</tbody>
</table>

Chapter 4. Debugging C/C++ routines  179
<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<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_KSDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS_PATH</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_KSDS_PATH</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_MODCB</td>
<td>Set when a low level VSAM MODCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_TESTCB</td>
<td>Set when a low level VSAM TESTCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_SHOWCB</td>
<td>Set when a low level VSAM SHOWCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_GENCB</td>
<td>Set when a low level VSAM GENCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_GET</td>
<td>Set when the last op was a low level VSAM GET; if the GET fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_PUT</td>
<td>Set when the last op was a low level VSAM PUT; if the PUT fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_POINT</td>
<td>Set when the last op was a low level VSAM POINT; if the POINT fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_ERASE</td>
<td>Set when the last op was a low level VSAM ERASE; if the ERASE fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_ENDREQ</td>
<td>Set when the last op was a low level VSAM ENDREQ; if the ENDREQ fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_CLOSE</td>
<td>Set when the last op was a low level VSAM CLOSE; if the CLOSE fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__QSAM_GET</td>
<td>__error is not set (if abend (errno == 92), __abend is set, otherwise if read error (errno == 66), look at __msg.</td>
</tr>
<tr>
<td>__QSAM_PUT</td>
<td>__error is not set (if abend (errno == 92), __abend is set, otherwise if write error (errno == 65), look at __msg.</td>
</tr>
<tr>
<td>__QSAM_TRUNC</td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td>__QSAM_FREEPOOL</td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td>__QSAM_CLOSE</td>
<td>Sets __error to result of OS CLOSE macro.</td>
</tr>
<tr>
<td>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__QSAM_OPEN</td>
<td>Sets __error to result of OS OPEN macro.</td>
</tr>
<tr>
<td>__CMS_OPEN</td>
<td>Sets __error to result of FSOPEN.</td>
</tr>
<tr>
<td>__CMS_CLOSE</td>
<td>Sets __error to result of FSCLOSE.</td>
</tr>
<tr>
<td>__CMS_READ</td>
<td>Sets __error to result of FSREAD.</td>
</tr>
<tr>
<td>__CMS_WRITE</td>
<td>Sets __error to result of FSWRITE.</td>
</tr>
<tr>
<td>__CMS_STATE</td>
<td>Sets __error to result of FSSTATE.</td>
</tr>
<tr>
<td>__CMS_ERASE</td>
<td>Sets __error to result of FSERASE.</td>
</tr>
<tr>
<td>__CMS_RENAME</td>
<td>Sets __error to result of CMS RENAME command.</td>
</tr>
<tr>
<td>__CMS_EXTRACT</td>
<td>Sets __error to result of DMS EXTRACT call.</td>
</tr>
<tr>
<td>__CMS_LINERD</td>
<td>Sets __error to result of LINERD macro.</td>
</tr>
<tr>
<td>__CMS LINEWRT</td>
<td>Sets __error to result of LINEWRT macro.</td>
</tr>
<tr>
<td>__CMS_QUERY</td>
<td>__error is not set.</td>
</tr>
<tr>
<td>__HSP_CREATE</td>
<td>Indicates last op was a DSPSERV CREATE to create a hiperspace for a hiperspace memory file. If CREATE fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__HSP_DELETE</td>
<td>Indicates last op was a DSPSERV DELETE to delete a hiperspace for a hiperspace memory file during termination. If DELETE fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__HSP_READ</td>
<td>Indicates last op was a HSPSERV READ from a hiperspace. If READ fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__HSP_WRITE</td>
<td>Indicates last op was a HSPSERV WRITE to a hiperspace. If WRITE fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__HSP_EXTEND</td>
<td>Indicates last op was a HSPSERV EXTEND during a write to a hiperspace. If EXTEND fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__CICS_WRITEQ_TD</td>
<td>Sets __error with error code from EXEC CICS WRITEQ TD.</td>
</tr>
<tr>
<td>__LFS_OPEN</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. Reason code from HFS services can be looked up in <a href="#">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
<tr>
<td>__LFS_CLOSE</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in <a href="#">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
<tr>
<td>__LFS_READ</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in <a href="#">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 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 26 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.

```
#include <stdio.h>
int main(void){
    FILE *fp;
    fp = fopen("myfile.dat", "w");
    if (fp == NULL)
        perror("fopen error");
}
```

Figure 26. 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++ run-time 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.

---

Table 35. __last_op values and diagnosis information (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__LFS_WRITE</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference</td>
</tr>
<tr>
<td>__LFS_LSEEK</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference</td>
</tr>
<tr>
<td>__LFS_FSTAT</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference</td>
</tr>
</tbody>
</table>
Figure 27 is an example of a routine using __errno2().

```
#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 27. Example of a routine using __errno2()

Figure 28 shows the output from the sample routine in Figure 27.

```
fopen() failed: EDC5129I No such file or directory. (errno2=0x05620062)
__errno2 = 05620062
```

Figure 28. Sample output of a routine using __errno2()

Figure 29 is an example of a routine using the environment variable _EDC_ADD_ERRNO2.

```
#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 29. Example of a routine using _EDC_ADD_ERRNO2

Figure 30 shows the sample output from the routine in Figure 29.

```
fopen() failed: EDC5129I No such file or directory.
```

Figure 30. Sample output of a routine using _EDC_ADD_ERRNO2

Figure 31 on page 184 is an example of a routine using __err2ad() in combination with __errno2().
Figure 32 shows the sample output from the routine shown in Figure 31.

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++ Run-Time 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 78.

**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.

   aa1 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>If you want the WSA base address for:</th>
<th>Locate the WSA base address in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>application code</td>
<td>the WSA address field in the Enclave Control Blocks section</td>
</tr>
<tr>
<td>a fetched module</td>
<td>the WSA address field of the Fetch() Information section for the fetch() function pointer for which you are interested</td>
</tr>
<tr>
<td>a DLL</td>
<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 `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.

**Notes:**

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 `cdump` or `CEE3DMP` is called, but it is written to a separate ddname called `CEESNAP`. For information about `cdump`, `CEE3DMP`, and enabling the `CEESNAP` ddname, see “Generating a Language Environment dump of a C/C++ routine” on page 192.

**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 185. 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 @STATIC (associated with the file where the static variable is located) in the Writable Static Map section of the prelinker map. Figure 33 on page 187 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.
   \[ X'02D66E40' + X'58' = X'2D66E98' \]

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).
   \[ X'2D66E98' + X'60' = X'2D66EF8' \]

When you are done, you have the address of the value of the static variable in the
Language Environment dump.

Figure 34 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 185. 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 35, the offset for DFHEIPTR is X'28'.

```
<table>
<thead>
<tr>
<th>Writable Static Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET LENGTH FILE ID INPUT NAME</td>
</tr>
<tr>
<td>0 1 00001 DFHC0011</td>
</tr>
<tr>
<td>4 1 00001 DFHC0010</td>
</tr>
<tr>
<td>8 2 00001 DFHDUMMY</td>
</tr>
<tr>
<td>C 2 00001 DFHB0025</td>
</tr>
<tr>
<td>10 2 00001 DFHB0024</td>
</tr>
<tr>
<td>14 2 00001 DFHB0023</td>
</tr>
<tr>
<td>18 2 00001 DFHB0022</td>
</tr>
<tr>
<td>1C 2 00001 DFHB0021</td>
</tr>
<tr>
<td>20 2 00001 DFHB0020</td>
</tr>
<tr>
<td>24 2 00001 DFHE180</td>
</tr>
<tr>
<td>28 4 00001 DFHEIPTR</td>
</tr>
<tr>
<td>2C 4 00001 DFHCPO11</td>
</tr>
<tr>
<td>30 4 00001 DFHCPO10</td>
</tr>
<tr>
<td>34 4 00001 DFHP025</td>
</tr>
<tr>
<td>38 4 00001 DFHP024</td>
</tr>
<tr>
<td>3C 4 00001 DFHP023</td>
</tr>
<tr>
<td>40 4 00001 DFHP022</td>
</tr>
<tr>
<td>44 4 00001 DFHP021</td>
</tr>
<tr>
<td>48 4 00001 DFHP020</td>
</tr>
<tr>
<td>4C 4 00001 DFHEICB</td>
</tr>
<tr>
<td>50 4 00001 DFHE109</td>
</tr>
<tr>
<td>54 4 00001 DFHLOVER</td>
</tr>
<tr>
<td>58 420 00001 @STATIC</td>
</tr>
</tbody>
</table>
```

Figure 35. 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 185. 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 36 shows how to locate NORENT C static variables by adding the Static Storage Area CSECT address to the variable offset.

---

**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 37 shows an example code for the parameter variable.

   ```c
   func0() {
       func1(a1, a2);
   }
   func1(int ppx, int pp0) {
   }
   ```

   **Figure 37. Example code for parameter variable**

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.

   ```plaintext
   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 189.

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 38 shows an example code for the parameter variable.

```c
func0() {
    
    func1(a1,a2);
    
}

func1(int ppx, int pp0) {
    
}
```

Figure 38. 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 39.

```plaintext
ppx  62-0:62  Class = parameter,  Location = 188(r13),  Length = 4
pp0  62-0:62  Class = parameter,  Location = 192(r13),  Length = 4
```

Figure 39. 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 40 on page 191 shows an example of a static aggregate.
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 42 on page 192] shows the Writable Static Map produced by the prelinker.

```c
static struct {
    short int ss01;
    char ss02[56];
    int ss03;
} ss0;
```

[Figure 41] shows an example aggregate map.
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 41 on page 191
   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++ Run-Time 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()
Figure 43 on page 195 shows a sample C routine that uses the cdump function to
generate a dump. Figure 48 on page 199 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);
    }

    printf(fp1, "record 1\n");
    printf(fp1, "record 2\n");
    printf(fp1, "record 3\n");

    fp2 = fopen("memory.data", "wb,type=memory");
    if (!fp2) {
        perror("Could not open memory.data for write");
        exit(102);
    }
```

*Figure 43. Example C routine using cdump() to generate a dump (Part 1 of 2)*
Figure 44. Fetched module for C routine

```c
#include <ctest.h>

int func1(void) {
    cdump("This is a sample dump");
    return(0);
}
```

**Figure 44. Fetched module for C routine**

Sample C++ routine that generates a Language Environment dump

*Figure 45 on page 197* shows a sample C++ routine that uses a protection exception to generate a dump.
```cpp
#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 45. Example C++ routine with protection exception generating a dump

```cpp
#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 46. Template file STACK.C

```cpp
#ifndef __STACK__
#define __STACK__
template <class T> class Stack {
public:
    Stack() {
        char* badPtr = 0; badPtr -= (0x01010101);
        head = (Node*) badPtr; /* head initialized to 0xFEFEFEFF */
    }
    T pop();
    void push(T);
private:
    struct Node {
        T value;
        struct Node* next;
    }* head;
#endif
```

Figure 47. Header file stack.h
Sample Language Environment dump with C/C++-specific information

The sample dump in Figure 48 on page 199 was produced by compiling the routine in Figure 43 on page 195 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 compile 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 204.
CEEDump V1 R12.0: This is a sample dump 03/07/10 1:12:11 PM Page: 1

ASID: 0041 Job ID: JOB12852 Job name: CSAMPLE Step name: STEP1 UserID: HEALY

CEE3845I CEEDump Processing started.
CEE3DMP called by program unit (entry point __cdump) at offset +0000017C.

Snap was unsuccessful

Registers on Entry to CEE3DMP:

PM....... 0100
GPR0..... 00000000_20E566A4 GPR1..... 00000000_20FCB5A0 GPR2..... 00000000_00000001 GPR3..... 00000000_20E56752
GPR4..... 00000000_20FC8588 GPR5..... 00000000_20FCB5C4 GPR6..... 00000000_00000014 GPR7..... 00000000_20902760
GPR8..... 00000000_00000030 GPR9..... 00000000_20FC7038 GPR10.... 00000000_00000091 GPR11.... 00000000_00000091
GPR12.... 00000000_209139B0 GPR13.... 00000000_20FCB508 GPR14.... ********_20E56896 GPR15.... 00000000_A0F0898
FP0...... 00000000_00000000 FPR2...... 18000000 00000000
FP4...... 00000000_00000000 FPR6...... 00000000 00000000

Information for enclave main

Information for thread 8000000000000000

Registers on Entry to CEE3DMP:

PM....... 0100
GPR0..... 00000000_20E566A4 GPR1..... 00000000_20FCB5A0 GPR2..... 00000000_00000001 GPR3..... 00000000_20E56752
GPR4..... 00000000_20FC8588 GPR5..... 00000000_20FCB5C4 GPR6..... 00000000_00000014 GPR7..... 00000000_20902760
GPR8..... 00000000_00000030 GPR9..... 00000000_20FC7038 GPR10.... 00000000_00000091 GPR11.... 00000000_00000091
GPR12.... 00000000_209139B0 GPR13.... 00000000_20FCB508 GPR14.... ********_20E56896 GPR15.... 00000000_A0F0898
FP0...... 00000000_00000000 FPR2...... 18000000 00000000
FP4...... 00000000_00000000 FPR6...... 00000000 00000000

Traceback:

DSA Entry E Offset Statement Load Mod Program Unit Service Status
1 __cdump +0000017C CEEEV003 D1908:e Call
2 func1 +0000006E 5 MODULE1 MODULE1 Call
3 @@FECBMODULE1 -002BD99E Call
4 @@GETFN +000000C2 CEEEV003 Call
5 main +00000392 64 CSAMPLE CSAMPLE Call
6 EDCZMINV +000000C2 CEEEV003 Call
7 CEEBBEXT +00000186 CEEBBEXT D1908 Call

DSA Addr E Addr PU Offset Comp Date Compile Attributes
1 20FCB508 20E56718 20E56718 +0000017C 20061215 LIBRARY EBCDIC HFP
2 20FCB468 20900310 20900310 +0000009E 19970314 C/C++
3 20FCB378 20FEB9F0 20FB9F0 -002B999E LIBRARY
4 20FCB2D8 20E3C0 20E3C0 +00000186 12/15/06 LIBRARY
5 20FCB208 20901078 20901078 +00000392 19970314 C/C++
6 20FCB0FD 206999E 206999E +000000C2 20061215 LIBRARY
7 20FCB030 20992208 20992208 +00000186 20061215 CEL

Fully Qualified Names

DSA Entry Program Unit Load Module
1 func1 POSIX.CRTL.C(MODULE1) MODULE1
2 main POSIX.CRTL.C(CSAMPLE) CSAMPLE

Parameters, Registers, and Variables for Active Routines:

main (DSA address 20FCB208):

UPSTACK DSA

Saved Registers:

GPR4..... A099220E GPR5..... 20FC7098 GPR6..... 20FCB30 GPR7..... 20FCB30
GPR8..... 00000000 GPR9..... 00000000 GPR10.... 00000000 GPR11.... 00000000
GPR12.... 209139B0 GPR13.... 20FCB508 GPR14.... ********_20E56896 GPR15.... 00000000_A0F0898

Local Variables:

fetchPtr signed int (*) (void) 0x20FB3E30
fp2 struct __ffile * 0x20FF5AFC
fp1 struct __ffile * 0x20FF4024
rc signed int 0

Figure 48. Example dump from sample C routine (Part 1 of 6)
Control Blocks for Active Routines:

DSA for func1: 20FCB468
+000000 FLAGS... 1000 member... 0000 BKC...... 20FCB378 FCW...... D4D6C4E4 R14...... A0900380
+000010 R15...... 20FCB500 R1....... 20FCB500 R2....... A090035E
+000024 R4....... 20FCB378 FWC...... D4D6C4E4 R14...... A0900380
+000030 R15...... 20FCB500 R1....... 20FCB500 R2....... A090035E
+000040 R4....... 20FCB378 FWC...... D4D6C4E4 R14...... A0900380
+000050 R15...... 20FCB500 R1....... 20FCB500 R2....... A090035E
+000060 R4....... 20FCB378 FWC...... D4D6C4E4 R14...... A0900380
+000070 R15...... 20FCB500 R1....... 20FCB500 R2....... A090035E
+000080 R4....... 20FCB378 FWC...... D4D6C4E4 R14...... A0900380

[1] Storage for Active Routines:

DSA frame: 20FCB468
+000000 20FCB468 10000000 20FCB378 D4D6C4E4 A0900380 20E56718 20FCB508 20FCB500 A0D2E5F2 |........MODU.....V...........KV2|
+000020 20FCB488 A090035E A09922EC 20FEBF70 20FEBE30 20902760 00000030 20FC7038 A0900310 |...;.r.............-............|
+000040 20FCB4A8 A0900310 209139B0 00000000 20FCB508 A099DD88 20FCB378 20FCB534 20900B08 |.....j...........r.h............|
+000060 20FCB4C8 20FCB378 20900990 00000000 20FCB390 00000000 00000000 20991D52 A0915770 |.........................r...j..|
+000080 20FCB4E8 A0991A28 209139B0 00000000 20FCB5A0 20C4D1F2 00000400 20FEBF70 00000009 |.r...j...........DJ2............|

[2] Control Blocks Associated with the Thread:

CAA: 209139B0
+000000 209139B0 00000800 00000000 20FCB018 20FEB018 00000000 00000000 00000000 00000000 |................................|
+000020 209139D0 00000000 00000000 20910A58 00000000 00000000 00000000 00000000 00000000 |.........j......................|
+000040 209139F0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |................................|
+000060 20913A10 00000000 00000000 00000000 00000000 00000000 00000000 80014608 00000000 |................................|
+000080 20913A30 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |................................|

[2A] C-C++ CAA information:

C-C++ Specific CTHD........ 2090E858
C-C++ Specific CEDB........ 2090F8D8
C-C++ Specific Thread block: 2090E858
+000000 2090E858 C3E3C8C4 00000380 2090E858 00000000 20D15090 00000000 00000000 00000000 |CTHD......Y......J&.............|
+000020 2090E878 00000000 00000000 000000C4 2090EDE0 00000000 00000000 00000000 00000000 |...............D................|
+000040 2090E898 20FEBF90 2090F9B0 2090F9C4 20FC71B8 2090F6BC 2090F28C 2090F4A4 20D1B242 |......9...9D......6...2...4u.J..|
+000060 2090E8B8 00004650 20910130 40400000 00000000 00100000 00000000 00000000 00000000 |...&.j.. ......................|
+000080 2090E8D8 7FFFFFFF FFFFFFFF 71FFFFFF FFFFFFFF 3C100000 00000000 34100000 00000000 |"...............................|
+000780 20910058 00000000 00000000 00000000 0001D100 00002E80 00000000 00000000 00000000 |..............J.................|
+0007A0 20910078 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |................................|

Figure 48. Example dump from sample C routine (Part 2 of 6)
errno value................. 0
memory file block chain.... 20FF5D18
open FCB chain............... 20FF5B10
GTAB table.................. 2090EBE0

Signal information:
SIGFPE:
    function pointer... 20901D20
    WSA address... A0FC7038
    function name... hsigfpe
SIGTERM:
    function pointer... 20901E90
    WSA address... A0FC7038
    function name... hsigterm

Enclave variables:
*.*.C(CSAMPLE):>hsigterm
    void () 0x20901E90
*.*.C(CSAMPLE):>hsigfpe
    void () 0x20901D20
*.*.C(CSAMPLE):>xcount
    signed int 0
*.*.C(CSAMPLE):>main
    signed int (void) 0x20901078
*.*.C(CSAMPLE):>atf1
    void (void) 0x20901FF8
*.*.C(CSAMPLE):>st2
    signed int 255
*.*.C(CSAMPLE):>st1
    signed int 99
*.*.C(MODULE1):>func1
    signed int (void) 0x20900310

Enclave Control Blocks:
EDB: 20912648
    +000000 20912648 C3C5C5C5 C4C24040 C0000001 20913870 20912D50 00000000 00000000 00000000 |CEEEDB .....j...j.&............|
    +000020 20912668 20912B58 20912B88 A0915770 20912198 00000000 00000000 20912768 00000000 |.j...j.h.j...j.q.........j......|
    +000040 20912688 00000000 00000000 00006F58 00000000 00000000 A0A33B58 2090A880 20FEBF60 |..........?..........t....y....-|
    +000060 209126A8 0000F460 00000000 20911E58 00000000 20914550 00000000 20AF4660 209139B0 |..4-.....j.......j.&.......-.j..|
    +000080 209126C8 50000000 0000FAF6 00000000 00000001 00000000 00000000 00006FF0 008FF4E8 |&......6..................?0..4Y|
    +0000A0 209126E8 00000001 00000100 2090A988 209126F0 00000000 00000000 00000000 00000001 |..........zh.j.0................|
MEML: 20913870
    +000000 20913870 00000000 00000000 209945B0 00000000 00000000 00000000 209945B0 00000000 |.........r...............r......|
    +000020 20913890 00000000 00000000 209945B0 00000000 2090F8D8 00000000 A0B07F78 00000000 |.........r........8Q......".....|
    +000040 209138B0 00000000 00000000 209945B0 00000000 00000000 00000000 209945B0 00000000 |.........r...............r......|
    +000060 209138D0 - +00011F 2091398F same as above

Atexit information:
function pointer... A0901FF8
WSA address................. 20FC7038
function name... atf1

Fetch information:
fetch pointer : 20FEBF90
function pointer... A0900310
WSA address... 20FC7038

Enclave Storage:
Initial (User) Heap : 20FEB018
    +000000 20FEB018 C3C5C5C5 C4C24040 00008000 00000000 00000000 00000000 00000000 00000000 |CEEEDB .....j...j.&............|
    +000020 20FEB038 20FEB018 00000120 20FEB160 20FEB348 20FEB385 20FEB3C2 20FEB3FF 20FEB43C |...........-.......e...B........|
    +000040 20FEB058 20FEB479 20FEB4B6 20FEB4F3 20FEB900 20FEB93D 00000000 00000000 00000000 |...........3....................|...
    +001080 20FEB98 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |................................|
    +0010A0 20FECB88 - +007FFF 20FF3017 same as above
LE/370 Anywhere Heap : 20FC7000
    +000000 20FC7000 C3C5C5C5 21007000 20912B58 20912B58 20912B58 20912B58 20912B58 20912B58 |HANC.j.....j.............h....|
    +000020 20FC7020 20FC7000 00001900 00000000 00000000 00000000 00000000 00000000 00000000 |..........2 func1 PO|
    +000040 20FC7040 2090F28C 2090F28C 00000000 00000000 00000000 00000000 00000000 00000000 |SIX.CRTL.C(MODULE1) |...|
    +000240 21007240 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |................................|
    +000260 21007260 - +00A5F 21008A5F same as above

Figure 48. Example dump from sample C routine (Part 3 of 6)
Figure 48. Example dump from sample C routine (Part 4 of 6)
Figure 48. Example dump from sample C routine (Part 5 of 6)
Finding C/C++ information in a Language Environment dump

When a Language Environment traceback or dump is generated for a C/C++ routine, information is provided that is unique to C/C++ routines. C/C++-specific information includes:

- Control block information for active routines
- Condition information for active routines
- Enclave level data

Each of the unique C/C++ sections of the Language Environment dump are described in Table 37.

Table 37. Contents of the C/C++ sections of the 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>
Table 37. Contents of the C/C++ sections of the Language Environment (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</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 run-time environment. For each C/C++ 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 C/C++ thread level. For more information about the MFCB, see “Memory File Control Block” on page 206.  
  **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” on page 206. |
| [3] Signal Information | When the POSIX(OFF) run-time option is specified, signal information is provided in the dump to aid you in debugging. For each signal that is disabled with SIG_IGN, an entry value of 00000001 is made in the first field of the Signal Information field for the specified signal name.  
  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 @@FECB is entered as the function name and the address of the fetched pointer is in the first field.  
  If you compile a C routine as NORENT, the WSA address is not available (N/A). For more information about the signal function, see z/OS XL C/C++ Programming Guide. |
| [4] WSA Address | 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. |
| [5] atexit() Information | Lists the functions registered with the atexit() function that would be run at normal termination. The functions are listed in chronological order of registration.  
  If you compile a C routine as NORENT, the WSA address is not available (N/A). For more information about the atexit() function, see z/OS XL C/C++ Run-Time Library Reference. |
| [6] fetch() Information | Shows information about modules that you have dynamically loaded using fetch(). For each module that was fetched, the fetch() pointer and the function pointer are included.  
  ptr1 = fetch("MOD");  
  If you compile a C routine as NORENT, the WSA address is not available (N/A). For more information about the fetch() function, see z/OS XL C/C++ Programming Guide. |
Table 37. Contents of the C/C++ sections of the Language Environment (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7] File Control Block Information</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>• Access method control block (ACB) address</td>
</tr>
<tr>
<td></td>
<td>• Data control block (DCB) address</td>
</tr>
<tr>
<td></td>
<td>• Data control block extension (DCBE) address</td>
</tr>
<tr>
<td></td>
<td>• Job file control block (JFCB) address</td>
</tr>
<tr>
<td></td>
<td>• RPL address</td>
</tr>
<tr>
<td></td>
<td>• Current buffer address</td>
</tr>
<tr>
<td></td>
<td>• Saved buffer address</td>
</tr>
<tr>
<td></td>
<td>• ddname</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.)</td>
</tr>
<tr>
<td></td>
<td>For information about the job file control block, see z/OS MVS Data Areas in z/OS Internet Library at <a href="http://www.ibm.com/systems/z/os/zos/bkserv/">http://www.ibm.com/systems/z/os/zos/bkserv/</a>.</td>
</tr>
<tr>
<td>[8] Information for __amrc</td>
<td>__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 &quot;Debugging C/C++ programs&quot; on page 175 and to z/OS XL C/C++ Programming Guide.</td>
</tr>
<tr>
<td>[9] Errno Information</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 49 on page 207.

#### 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 50] 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.
Parameters, Registers, and Variables for Active Routines:

```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR0</td>
<td>183F6CC0</td>
</tr>
<tr>
<td>GPR1</td>
<td>00021278</td>
</tr>
<tr>
<td>GPR2</td>
<td>183F6968</td>
</tr>
<tr>
<td>GPR3</td>
<td>17F02408</td>
</tr>
<tr>
<td>GPR4</td>
<td>000000F8</td>
</tr>
<tr>
<td>GPR5</td>
<td>80000000</td>
</tr>
<tr>
<td>GPR6</td>
<td>98125022</td>
</tr>
<tr>
<td>GPR7</td>
<td>80007F98</td>
</tr>
<tr>
<td>GPR8</td>
<td>000212F0</td>
</tr>
<tr>
<td>GPR9</td>
<td>80000000</td>
</tr>
<tr>
<td>GPR10</td>
<td>98125022</td>
</tr>
<tr>
<td>GPR11</td>
<td>80007F98</td>
</tr>
<tr>
<td>GPR12</td>
<td>00015920</td>
</tr>
<tr>
<td>GPR13</td>
<td>000213B0</td>
</tr>
<tr>
<td>GPR14</td>
<td>97F01E1E</td>
</tr>
<tr>
<td>GPR15</td>
<td>0000002F</td>
</tr>
<tr>
<td>FPR8</td>
<td>3FF33333</td>
</tr>
<tr>
<td>FPR9</td>
<td>3FF33333</td>
</tr>
<tr>
<td>FPR10</td>
<td>3FF33333</td>
</tr>
<tr>
<td>FPR11</td>
<td>3FF33333</td>
</tr>
<tr>
<td>FPR12</td>
<td>40260000</td>
</tr>
<tr>
<td>FPR13</td>
<td>40220000</td>
</tr>
<tr>
<td>FPR14</td>
<td>40260000</td>
</tr>
<tr>
<td>FPR15</td>
<td>40220000</td>
</tr>
<tr>
<td>GPRG</td>
<td>00000000</td>
</tr>
<tr>
<td>Storage</td>
<td>17F01E32</td>
</tr>
</tbody>
</table>
```

Figure 51. Parameters, registers, and variables for active routines

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 52 shows sample output.

Condition Information for Active Routines
Condition Information for ./celsamp.c (DSA address 000213B0)
CIB Address: 00021F90
Current Condition:
CEE3224S The system detected an IEEE division-by-zero exception.
Location:
Program Unit: ./celsamp.c
Program Unit: Entry: goo Statement: 78 Offset: +000000BA
Machine State:
ILC..... 0004 Interruption Code..... 0007
PSW..... 078D0400 97F01E46
GPR0..... 183F6CC0 GPR1..... 00021278 GPR2..... 183F6968 GPR3..... 17F02408
GPR4..... 000000F8 GPR5..... 80000000 GPR6..... 98125022 GPR7..... 80007F98
GPR8..... 000212F0 GPR9..... 80000000 GPR10.... 98125022 GPR11.... 80007F98
GPR12.... 00015920 GPR13.... 000213B0 GPR14.... 97F01E1E GPR15.... 0000002F
FPC...... 40084000
FPR8..... 3FF33333 33333333
FPR9..... 3FF33333 33333333
FPR10.... 3FF33333 33333333
FPR11.... 3FF33333 33333333
FPR12.... 40260000 00000000
FPR13.... 40220000 00000000
FPR14.... 40260000 00000000
FPR15.... 40220000 00000000
Storage dump near condition, beginning at location: 17F01E32
+000000 17F01E32 68201008 5810D0F0 68401010 B31B0024 B31D0002 B3050000 582000F4 584031C2 &Lv;.......0. .................4. .&Lv;

Figure 52. Condition information for active routines

Sample Language Environment dump with XPLINK-specific information

The programs tranmain (Figure 53 on page 209) and trandll (Figure 54 on page 209) were used to produce a Language Environment dump. The Language Environment dump produced by running these program is shown in Figure 55 on page 210. 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 sideedge from the other. Explanations for some of the sections are in “Finding XPLINK information in a Language Environment dump” on page 212.
```c
#include <stdio.h>
#include <ctest.h>
#include <leawi.h>
#pragma export(tran1)
#pragma export(tran3)

int tran1(int parm1, int parm2, int parm3, long double parm4, int parm5) {

    printf("Tran1: Call Tran2\n");
    retval = tran2(parm1, parm2, parm3, parm4, parm5);
    printf("Tran1: Return value from Tran2 = \%d\n", retval);
    return retval;
}

int tran2(int parm1, int parm2, int parm3, long double parm4, int parm5) {

    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 53. Sample XPLINK-compiled program (tranmain) which calls a NOXPLINK-compiled program**

```c
#include <stdio.h>
#pragma runopts(TRACE(ON,IM,NODUMP,LE=1),XPLINK(ON),TERMDHDCT(UADUMP))
#include <stdio.h>
#pragma export(tran2)

int tran1(int, int, int, long double, int);
int tran3(int, 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);
}
```

**Figure 54. Sample NOXPLINK-compiled program (trandll) which calls an XPLINK-compiled program**
Information for enclave main

### Traceback:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0004030</td>
<td>CEEPLPKA</td>
<td>CEEHDSP</td>
<td>CEEHDSP</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>02</td>
<td>000024C</td>
<td>CEEPLPKA</td>
<td>CEEABD0</td>
<td>CEEABD0</td>
<td>01908</td>
<td>Exception</td>
</tr>
<tr>
<td>03</td>
<td>0000074</td>
<td>CEEPLPKA</td>
<td>CEEABD0</td>
<td>CEEABD0</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>04</td>
<td>0000006</td>
<td>26</td>
<td>XNTDLL</td>
<td>tran1.c</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>05</td>
<td>0001026</td>
<td>CEEPLPKA</td>
<td>CEEVRONU</td>
<td>CEEVRONU</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>06</td>
<td>0000070</td>
<td>27</td>
<td>XNTRAN</td>
<td>tran1.c</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>07</td>
<td>00011FA</td>
<td>CEEPLPKA</td>
<td>CEEVRONU</td>
<td>CEEVRONU</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>08</td>
<td>000000F</td>
<td>14</td>
<td>XNTDLL</td>
<td>tran1.c</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>09</td>
<td>0001026</td>
<td>CEEPLPKA</td>
<td>CEEVRONU</td>
<td>CEEVRONU</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>10</td>
<td>000008C</td>
<td>18</td>
<td>XNTRAN</td>
<td>tran1.c</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>11</td>
<td>00011FA</td>
<td>CEEPLPKA</td>
<td>CEEVRONU</td>
<td>CEEVRONU</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>12</td>
<td>0000008</td>
<td>43</td>
<td>CELHNV03</td>
<td>EDCZHINV</td>
<td>01908</td>
<td>Cal1</td>
</tr>
<tr>
<td>13</td>
<td>000001B</td>
<td>CEEPLPKA</td>
<td>CEEBECKT</td>
<td>CEEBECKT</td>
<td>01908</td>
<td>Cal1</td>
</tr>
</tbody>
</table>

### Condition Information for Active Routines

#### Condition Information for CEL4ABD0 (DSA address 211OCA60)

- **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:**
    - ILA...... 0000
    - USM...... 0000
    - PSW...... 0000
    - GPR0..... 0000
    - GPR1..... 0000
    - GPR2..... 0000
    - GPR3..... 0000
    - GPR4..... 0000
    - GPR5..... 0000
    - GPR6..... 0000
    - GPR7..... 0000
    - GPR8..... 0000
    - GPR9..... 0000
    - GPR10..... 0000
    - GPR11..... 0000
    - GPR12..... 0000
    - GPR13..... 0000
    - GPR14..... 0000
    - GPR15..... 0000
  - **ABEND code:** 000003E9 Reason code: 00000000
  - **Storage dump near condition, beginning at location:** 20AFCC44

#### Full Qualified Names

<table>
<thead>
<tr>
<th>Entry</th>
<th>Program Unit</th>
<th>Load Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>./trandll.c</td>
<td>XNTDLL</td>
</tr>
<tr>
<td>02</td>
<td>./trandll.c</td>
<td>XNTDLL</td>
</tr>
<tr>
<td>03</td>
<td>./trandll.c</td>
<td>XNTDLL</td>
</tr>
<tr>
<td>04</td>
<td>./trandll.c</td>
<td>XNTDLL</td>
</tr>
<tr>
<td>05</td>
<td>./trandll.c</td>
<td>XNTDLL</td>
</tr>
<tr>
<td>06</td>
<td>./tranmain.c</td>
<td>XNTRAN</td>
</tr>
<tr>
<td>07</td>
<td>./tranmain.c</td>
<td>XNTRAN</td>
</tr>
<tr>
<td>08</td>
<td>./tranmain.c</td>
<td>XNTRAN</td>
</tr>
<tr>
<td>09</td>
<td>./tranmain.c</td>
<td>XNTRAN</td>
</tr>
</tbody>
</table>

---

Figure 55. Example dump of calling between XPLINK and non-XPLINK programs (Part 1 of 3)
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(.................|
    - Storage around GPR15 (209BFCD0)
  - (-0020 209BFCB0 209BFA00 F2F0F0F6 F1F2F1F5 F1F1F5F2 F0F0F0F1 F0F9F0F0 0005C4F1 F9F0F800 |....20061215115200010900..D1908.|
  - (+0000 209BFCD0 47F0F014 00C3C5C5 00000098 000000E0 47F0F001 90ECD00C 18BF5800 B0D05810 |.00..CEE...q.....00.............|
  - (+0020 209BFCF0 D04C1E01 5500C00C 47D0B034 58F0C2BC 05EF181F 5000104C D7011000 100018FD |.<...........0B.....&..<P.......|

- **Local Variables:**
  - `timing` signed long int 1
  - `code` signed long int 1001

**CEEVRONU (DSA address 2110C750):**

- **TRANSITION DSA**

  Saved Registers:
  - GPR0..... 20914D70
  - GPR1..... 212B6D70
  - GPR2..... 212BBE18
  - GPR3..... 0000001F
  - GPR4..... 212B6530
  - GPR5..... 21109718
  - GPR6..... 00000000
  - GPR7..... 00000000
  - GPR8..... A090003
  - GPR9..... 212BB868
  - GPR10.... 212BB8C0
  - GPR11.... 20ACDF1C
  - GPR12.... 209139B0
  - GPR13.... 2110C750
  - GPR14.... A0ACDA80
  - GPR15.... 212BB8C0

- **tran2 (DSA address 212B86530):**

  - **DOWNSTACK 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..... 2110C598
  - GPR2..... 55555555
  - GPR3..... 212BBA7A
  - GPR4..... 21286530
  - GPR5..... 21109718
  - GPR6..... 00000000
  - GPR7..... 00000000
  - GPR8..... 209000B8
  - GPR9..... 209000B8
  - GPR10.... 209000B8
  - GPR11.... 209000B8
  - GPR12.... 209000B8
  - GPR13.... 2110C5CB
  - GPR14.... 209000B8
  - GPR15.... 0000000C

- **Local Variables:**
  - `retval` signed int -455613482

**CEEVROND (DSA address 212B86580):**

- **TRANSITION DSA**

  Saved Registers:
  - GPR0..... ********
  - GPR1..... ********
  - GPR2..... ********
  - GPR3..... ********
  - GPR4..... 212B65B0
  - GPR5..... 20914E80
  - GPR6..... 209000E8
  - GPR7..... 209000E8
  - GPR8..... 209000E8
  - GPR9..... 209000E8
  - GPR10.... ********
  - GPR11.... ********
  - GPR12.... ********
  - GPR13.... ********
  - GPR14.... ********
  - GPR15.... ********

- **tran1 (DSA address 2110C500):**

  - **UPSTACK DSA**

  Saved Registers:
  - GPR0..... 211080A8
  - GPR1..... 2110C598
  - GPR2..... 55555555
  - GPR3..... 21288A7A
  - GPR4..... 21286530
  - GPR5..... 21109718
  - GPR6..... 00000000
  - GPR7..... 00000000
  - GPR8..... A090002B
  - GPR9..... 21288C8
  - GPR10.... 21288A40
  - GPR11.... 20ACDF1C
  - GPR12.... 209139B0
  - GPR13.... 2110C5B0
  - GPR14.... A12BB834
  - GPR15.... A0ACAA48

---

Figure 55. Example dump of calling between XPLINK and non-XPLINK programs (Part 2 of 3)
Finding XPLINK information in a Language Environment dump

Table 38 describes the specific XPLINK information in sections of the 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>
### Section 2 Parameters, Registers, and Variables for Active Routines

In this section, each DSA is identified as one of the following:

**UPSTACK DSA**

The DSA format is that for a NOXPLINK-compiled program that uses an upward growing stack.

**DOWNSTACK DSA**

The DSA format is that for an XPLINK-compiled program that uses an downward growing stack.

**TRANSITION DSA**

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).

### Section 3 Control Blocks for Active Routines

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.

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.)

---

### 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 163.

- **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`:
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.

```
NameOfCallingFunction
---->{xxx} NameOfCalledFunction{(input_parameters)}
```

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 run-time 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.

Figure 56 on page 216 shows a non-XPLINK trace that has examples of C/C++ trace table entry types 1 thru 4.
Figure 56. Trace table with C/C++ trace table entry types 1 thru 4 (Part 1 of 2)
Figure 57 on page 218 shows an XPLINK trace that has examples of the trace entries 5 and 6.
Most recent trace entry is at displacement: 000D80

<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>22.41.35.433944</td>
<td>26C7000000000000</td>
</tr>
<tr>
<td>+000000</td>
<td>22.41.35.433948</td>
<td>26C7000000000000</td>
</tr>
<tr>
<td>+0000000</td>
<td>22.41.35.433952</td>
<td>26C7000000000000</td>
</tr>
<tr>
<td>+00000000</td>
<td>22.41.35.433957</td>
<td>26C7000000000000</td>
</tr>
<tr>
<td>+000000000</td>
<td>22.41.35.433963</td>
<td>26C7000000000000</td>
</tr>
<tr>
<td>+0000000000</td>
<td>22.41.35.433967</td>
<td>26C7000000000000</td>
</tr>
<tr>
<td>+00000000000</td>
<td>22.41.35.433972</td>
<td>26C7000000000000</td>
</tr>
<tr>
<td>+000000000000</td>
<td>22.41.35.433974</td>
<td>26C7000000000000</td>
</tr>
</tbody>
</table>

Figure 57. Trace table with XPLINK trace table entries 5 and 6

Figure 58 on page 219 shows an example of the format of the trace table entry type 7 and 8.
Figure 59 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: 000800

<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 22.10.56.799195</td>
<td>Date 2005.05.01</td>
</tr>
<tr>
<td>+000010</td>
<td>Member ID... 01 Flags......</td>
<td>Entry Type.............</td>
</tr>
<tr>
<td>+000018</td>
<td>C35C5C8 E5F0F0F3</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+000038</td>
<td>40404040 40404040 40404040</td>
<td>7AC5C4C3 E9C8C9D5</td>
</tr>
<tr>
<td>+000058</td>
<td>40404040 7A948189 956909581</td>
<td>956909581 956909581</td>
</tr>
<tr>
<td>+000078</td>
<td>40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000080</td>
<td>Time 22.10.56.804695</td>
<td>Date 2005.05.01</td>
</tr>
<tr>
<td>+000090</td>
<td>Member ID... 01 Flags......</td>
<td>Entry Type.............</td>
</tr>
<tr>
<td>+000098</td>
<td>C35C5C57 D3070C21 40404040</td>
<td>40404040 7AC5C4C3 E9C8C9D5</td>
</tr>
<tr>
<td>+000088</td>
<td>40404040 40404040 40404040</td>
<td>7AC5C4C3 D3070C21</td>
</tr>
<tr>
<td>+000068</td>
<td>40404040 7AC3C5C5 D7E3D3D6</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+0000F8</td>
<td>40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000100</td>
<td>Time 22.10.56.825094</td>
<td>Date 2005.05.01</td>
</tr>
<tr>
<td>+000110</td>
<td>Member ID... 01 Flags......</td>
<td>Entry Type.............</td>
</tr>
<tr>
<td>+000118</td>
<td>B1FB95F9 83F4F1A7 40404040</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+000138</td>
<td>40404040 40404040 40404040</td>
<td>40404040 7A948189</td>
</tr>
<tr>
<td>+000158</td>
<td>40404040 7A6A495 83A38996</td>
<td>956909581 956909581</td>
</tr>
<tr>
<td>+000178</td>
<td>40404040 6DA3966D F0404040</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+000188</td>
<td>40404040 40404040 40404040</td>
<td>40404040 7A404040</td>
</tr>
<tr>
<td>+000198</td>
<td>40404040 40404040 40404040</td>
<td>40404040 7A404040</td>
</tr>
<tr>
<td>+000218</td>
<td>B1FB95F9 83F4F1A6 83A38996</td>
<td>956909581 956909581</td>
</tr>
<tr>
<td>+000238</td>
<td>40404040 40404040 40404040</td>
<td>40404040 7A948189</td>
</tr>
<tr>
<td>+000258</td>
<td>40404040 7A979999 95638640</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+000278</td>
<td>40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000288</td>
<td>40404040 40404040 40404040</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+000300</td>
<td>Time 22.10.56.826629</td>
<td>Date 2005.05.01</td>
</tr>
<tr>
<td>+000310</td>
<td>Member ID... 01 Flags......</td>
<td>Entry Type.............</td>
</tr>
<tr>
<td>+000338</td>
<td>859587A3 86608598 40404040</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+000358</td>
<td>40404040 7A979999 95638640</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+000378</td>
<td>40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000388</td>
<td>40404040 40404040 40404040</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+000398</td>
<td>40404040 40404040 40404040</td>
<td>40404040 40404040</td>
</tr>
<tr>
<td>+000418</td>
<td>40404040 40404040 40404040</td>
<td>40404040 40404040</td>
</tr>
</tbody>
</table>
**Figure 59. Trace table with trace table entry type 20 (Part 2 of 2)**

<table>
<thead>
<tr>
<th>Time</th>
<th>Member ID</th>
<th>Flags</th>
<th>Entry Type</th>
<th>Additional Language Specific Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.10.56.826697</td>
<td>03</td>
<td>000000</td>
<td>00000008</td>
<td>errno information: Errno 0 Errmajr 00000000</td>
</tr>
<tr>
<td>0004B8</td>
<td>81F8F5F9</td>
<td>83F4F186</td>
<td>40404040 7AF6A495 83A38640 956D9581 94856D93</td>
<td>a859c41fnl34 :function_name_l</td>
</tr>
<tr>
<td>0004D8</td>
<td>40404040 40404040 40404040 40404040 :printf 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000558</td>
<td>81F8F5F9</td>
<td>83F4F186</td>
<td>9593F3F4 40404040 7AF6A495 83A38640 956D9581 94856D93</td>
<td>a859c41fnl35 :function_name_length_equal3</td>
</tr>
<tr>
<td>0005D8</td>
<td>40404040 40404040 40404040 40404040 :ond 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0006B8</td>
<td>40404040 40404040 40404040 40404040 :printf 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000758</td>
<td>40404040 40404040 40404040 40404040 :ond 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000858</td>
<td>40404040 40404040 40404040 40404040 :printf 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000968</td>
<td>40404040 40404040 40404040 40404040 :ond 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000A78</td>
<td>40404040 40404040 40404040 40404040 :printf 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional Language Specific Information:
errno information:
Thread Id .... 210CE8300000000 Ernmo ...... 0 Ernmajr .... 00000000
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 60 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 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.

```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 60. C routine with a divide-by-zero error

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 Run-Time 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 61 on page 222 shows the generated traceback from the dump.
2. Locate the instruction with the divide-by-zero error in the Pseudo Assembly Listing in Figure 62 on page 223.

The offset (within funcb) 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.
3. 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. For more information about finding automatic variables in a dump, see “Steps for finding automatic variables” on page 185.

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'20914F50'. Figure 63 shows the WSA address.

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 statint in the Writable Static Map in Figure 64. In this example, the offset is X'0'.

5. Add the WSA address of X'20914F50' to the offset of statint. The result is X'20914F50'. This is the address of the variable 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++ run-time 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 65 on page 225.

6. To find the variable statint in the writable static area, locate the closest address listed that is before the address of statint. In this case, that address is X'20914F50'. Count across X'00' to location X'20914F50'. The value at that location is X'49' (that is, statint is 73), and hence the fixed point divide exception.
Calling a nonexistent non-XPLINK function

Figure 66 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 67 on page 226. 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 Run-Time 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 $X'+000005A'$ 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.
2. Find the branch instructions at offset X'+0000005A' of funca in the listing in Figure 68 on page 227. 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 69, as produced by the binder.

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 70 on page 228 shows the sections of the dump.
Calling a nonexistent XPLINK function

Figure 71 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 Figure 72 on page 229. 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 Run-Time 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.
Condition processing resulted in the unhandled condition.

Information for enclave main

Traceback:

<table>
<thead>
<tr>
<th>DSA Addr</th>
<th>E Addr</th>
<th>F Addr</th>
<th>PU Addr</th>
<th>PU Offst</th>
<th>Comp Date</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2110C500</td>
<td>20902B08</td>
<td>20902B08</td>
<td>+000004030</td>
<td>20061215</td>
<td>CEL</td>
<td></td>
</tr>
<tr>
<td>2110C340</td>
<td>209E0B08</td>
<td>209E0B08</td>
<td>+00000092</td>
<td>20061215</td>
<td>CEL</td>
<td></td>
</tr>
<tr>
<td>211B5620</td>
<td>20900158</td>
<td>20900158</td>
<td>-20900158</td>
<td>20000404</td>
<td>C/C++ XPLINK EBCDIC HFP</td>
<td></td>
</tr>
<tr>
<td>211B56A0</td>
<td>209000D0</td>
<td>209000D0</td>
<td>+00000012</td>
<td>20000404</td>
<td>C/C++ XPLINK EBCDIC HFP</td>
<td></td>
</tr>
<tr>
<td>211B5A10</td>
<td>20CA4188</td>
<td>20CA4188</td>
<td>+00001252</td>
<td>20061215</td>
<td>CEL XPLINK EBCDIC HFP</td>
<td></td>
</tr>
<tr>
<td>211B5C60</td>
<td>20C6C6E8</td>
<td>20C6C6E8</td>
<td>+000000B4</td>
<td>20061214</td>
<td>LIBRARY</td>
<td></td>
</tr>
<tr>
<td>211B5620</td>
<td>209139B0</td>
<td>209139B0</td>
<td>+00000018</td>
<td>20061215</td>
<td>CEL</td>
<td></td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Condition Information for (DSA address 211B5620)

Current Condition:
CEE0198S The termination of a thread was signaled due to an unhandled condition.

Original Condition:
CEE3201S The system detected an operation exception (System Completion Code=0C1).

Location:

Program Unit: Entry: funca Statement: Offset: -20900158
Possible Bad Branch: Statement: Offset: +0000001C

Machine State:

ICL..... 0002 Interruption Code..... 0001
PSW..... 07BD2400 80000002
GPR0..... 2110C500 GPR1..... 2110C958 GPR2..... 20912648 GPR3..... 00000080
GPR4..... 209D7734 GPR5..... 20915000 GPR6..... 2110F620 GPR7..... 2110CE20
GPR8..... 2090C2A8 GPR9..... 2110E4FE GPR10.... 2110D4FF GPR11.... 209D2B08
GPR12.... 209139B0 GPR13.... 2110C500 GPR14.... 209D7734 GPR15.... 20900158

Storage dump near condition, beginning at location: 00000000

Figure 72. Sections of the dump from example C routine (calling a nonexistent XPLINK function) (Part 1 of 2)
2. Find the branch instruction at offset X'+0000001C' of funca in the listing in Figure 73. This instruction is BASR r7,r6. 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 74 on page 231.
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 74 shows the sections of the dump.

```
CLASS C_WSA LENGTH = 3C ATTRIBUTES = MRG, DEFER, RMODE=ANY
OFFSET = 0 IN SEGMENT 002 ALIGN = DBLWORD
```

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OFFSET</th>
<th>NAME</th>
<th>TYPE</th>
<th>LENGTH</th>
<th>SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>$PRIV000011</td>
<td>PART</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>exist</td>
<td>PART</td>
<td>28</td>
<td>EXIST</td>
</tr>
<tr>
<td></td>
<td>38</td>
<td>func_ptr</td>
<td>PART</td>
<td>4</td>
<td>func_ptr</td>
</tr>
</tbody>
</table>

Figure 74. Writable static map (calling a nonexistent XPLINK function)

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 SYMDUMP DD allocation information is not inherited. Even though the SYMDUMP allocation is not inherited, a SYMDUMP allocation must exist in the parent in order to obtain a HFS storage dump.

Alternatively, you can specify the DYNDUMP run-time 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 83.

```
/directory/coredump.pid
```
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 URCOMP.JRUSL.COREDUMP dump data set:

```
-------------------------- DATA SET INFORMATION ----------------------
Command ===>             
Data Set Name . . . : URCOMP.JRUSL.COREDUMP

General Data            Current Allocation
Management class . . : STANDARD  Allocated cylinders : 30
Storage class . . . : 05390    Allocated extents . : 1
Volume serial . . . : DPXDU1
Device type . . . . : 3380
Data class . . . . : 
Organization . . . : PS        Current Utilization
Record format . . . : FB          Used cylinders . . : 0
Record length . . . : 4160         Used extents . . : 0
Block size . . . . : 4160
1st extent cylinders: 30
Secondary cylinders : 10
Data set name type : 
Creation date . . . : 2001/08/30
Expiration date . . : ***None***
F1=Help    F2=Split   F3=End   F4=Return   F5=Rfind   F6=Rchange
F7=Up      F8=Down    F9=Swap  F10=Left    F11=Right  F12=Cancel

Figure 76. IPCS panel for entering data set information
```

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 84 for information about formatting Language Environment control blocks.

**Multithreading consideration**

Certain control blocks are locked while a dump is in progress. For example, a c.snap() 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) run-time 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 HEAPPOOLS statistics

To request a Language Environment storage report set RPTSTG(ON). If the C/C++ application specified the HEAPPOOLS(ON) run-time option, then the storage report displays HEAPPOOLS statistics. Figure 3 on page 14 is a sample storage report that shows HEAPPOOLS statistics for a multithreaded C/C++ application. The following sections describe the C/C++ specific heap pool information.

HEAPPOOLS storage statistics

The HEAPPOOLS run-time 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 run-time 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 HeapPool 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 run-time 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 run-time 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 HeapPool 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:

\[ \left( \frac{\text{Maximum Cells Used} \times \text{Element Size} \times 100}{\text{Initial Heap Size}} \right) \]

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 the [z/OS Language Environment Programming Guide](https://www.ibm.com/support/knowledgecenter/SSSLTB82_1.1.0/com.ibm.zos.v2r13.bmmdc0/bmmdc0c.htm).

**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

\[ p \quad \text{the number of the pool} \]

\[ ssss \quad \text{the cell size specified for the pool.} \]

• Successful Get Heap requests: xxxx-yyyy n

\[ xxxx \quad \text{the low side of the range} \]

\[ yyyy \quad \text{the high side of the range} \]
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:

\[
\text{Initial Heap Size} \times \left(\frac{\text{Extent Percent}}{100}\right) \div \left(8 + \text{Cell Size}\right)
\]

- 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:

\[
\left(\frac{\text{Maximum Cells Used} \times \left(\text{Cell Size} + 8\right) \times 100}{\text{Initial Heap Size}}\right)
\]

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.

Figure 77 on page 236 shows a sample storage report generated by __uheapreport(). For more information on the __uheapreport() function, see z/OS XL C/C++ Run-Time Library Reference. For tuning tips, see z/OS Language Environment Programming.
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 run-time 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 run-time option, the file specified by the _CEE_ENVFILE (or _CEE_ENVFILE_S) environment variable, or using the export command from the USS shell before any user code gets control (prior to the HLL user exit, static constructors, 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 USS 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 Trace Report (Figure 78) 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 CEEPRTN
  Called from: 257f6888 +000002b0 _term
  Called from: 05d46788 +0000040c (unknown)

DEALLOCATE of storage at 0x25a2e0c8
- sequence 11
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPRTN
  Called from: 257f6888 +000001bc _term
  Called from: 05d46788 +0000040c (unknown)

DEALLOCATE of storage at 0x25a2ecf8
- sequence 10
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPRTN
  Called from: 25601ae8 +000000b2 function3
  Called from: 25601bb8 +0000008c function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000bca main

ALLOCATE of storage at 0x25a2ecf8 for 5 bytes
- sequence 9
  Called from: 25a44330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPRTN
  Called from: 25601ae8 +000000b4 function3
  Called from: 25601bb8 +0000008c 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 CEEPRTN
  Called from: 25601ae8 +0000007e function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000062 main

DEALLOCATE of storage at 0x25a2ecd8
- sequence 7
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPRTN
  Called from: 25601ae6 +000000bc function1
  Called from: 25601a60 +00000062 main

DEALLOCATE of storage at 0x25a2ecd8
- sequence 6
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPRTN
  Called from: 25601ae6 +0000009e function1
  Called from: 25601a60 +00000062 main

Figure 78. Trace report generated by MEMCHECK VHM (Part 1 of 2)
The Heap Leak Report (Figure 79 on page 241) 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.
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.
- **(noname)** 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 documents for your level of COBOL:

- Enterprise COBOL for z/OS, V4R2, Programming Guide, SC23-8529
- Enterprise COBOL for z/OS, V3R4, Programming Guide, SC27-1412
- COBOL for OS/390 & VM Programming Guide, SC26-9049
- Enterprise COBOL for z/OS, V4R2, Language Reference, SC23-8528
- Enterprise COBOL for z/OS, V3R4, Language Reference, SC27-1408
- COBOL for OS/390 & VM Language Reference, SC26-9046

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:
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 run-time option are mutually exclusive, with DEBUG taking precedence.

Debugging lines must have a D in position 7 to identify them.

To use debugging lines and statements in your program, you must include both:
- WITH DEBUGGING MODE in the SOURCE-COMPUTER paragraph in the ENVIRONMENT DIVISION
- The DEBUG run-time option

Figure 80 on page 246 shows how to use the DISPLAY statement and the USE FOR DEBUGGING declarative to debug a program.
In the example in Figure 80, 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:

```
Trace For Procedure-Name : 501-Some-Routine
```

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:

```plaintext
USE FOR DEBUGGING ON ALL PROCEDURES.
```

Figure 80. Example of using the WITH DEBUGGING MODE clause
Using COBOL listings

When you are debugging, you can use one or more of the listings shown in Table 39. The following sections give an overview of each of these listings and the compiler option you use to obtain each listing.

Table 39. 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 81 on page 248 and Figure 82 on page 249 generate Language Environment dumps with COBOL-specific information.

COBOL program that calls another COBOL program

In Figure 81 on page 248 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 82 on page 249, 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 in Figure 83 on page 250. 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 251.
Figure 83. Sections of the Language Environment dump called from COBDUMP2 (Part 1 of 2)
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 84 on page 252 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 compile 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. 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.
Storage for each active routine

The Storage for Active Routines section of the dump, shown in Figure 85 on page 255, 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.
**Figure 85. Storage for active COBOL programs (Part 1 of 2)**

### Storage for Active Routines:

**COBUMP2:**
- Contents of base locators for files are:
  - 0-0001E038
- Contents of base locators for WORKING-STORAGE are:
  - 0-114A410B
- Contents of base locators for the LINKAGE SECTION are:
  - 0-00000000 1-114A4120

No indexes were used in this program.

No variably-located areas were used in this program.

No EXTERNAL data was used in this program.

No OBJECT instance data were used in this program.

No LOCAL-STORAGE was used in this program.

No DSA indexes were used in this program.

No FACTORY data was used in this program.

No XML data was used in this program.

### File record contents for COBUMP2

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0001E038</td>
</tr>
<tr>
<td>000200</td>
<td>0001E058 + 00003F 001107</td>
</tr>
</tbody>
</table>

### WORKING-STORAGE for COBUMP2

<table>
<thead>
<tr>
<th>Base Locators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>114A410B</td>
</tr>
<tr>
<td>000020</td>
<td>114A410B</td>
</tr>
<tr>
<td>000040</td>
<td>114A4218</td>
</tr>
<tr>
<td>000060</td>
<td>114A4238</td>
</tr>
<tr>
<td>000080</td>
<td>114A4258</td>
</tr>
<tr>
<td>000100</td>
<td>114A4278</td>
</tr>
<tr>
<td>000120</td>
<td>114A4298</td>
</tr>
<tr>
<td>000140</td>
<td>114A4318</td>
</tr>
<tr>
<td>000160</td>
<td>114A4338</td>
</tr>
</tbody>
</table>

### LINKAGE SECTION for COBUMP2

<table>
<thead>
<tr>
<th>Base Locators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>114A410B</td>
</tr>
<tr>
<td>000020</td>
<td>114A410B</td>
</tr>
<tr>
<td>000040</td>
<td>114A4218</td>
</tr>
<tr>
<td>000060</td>
<td>114A4238</td>
</tr>
<tr>
<td>000080</td>
<td>114A4258</td>
</tr>
<tr>
<td>000100</td>
<td>114A4278</td>
</tr>
<tr>
<td>000120</td>
<td>114A4298</td>
</tr>
<tr>
<td>000140</td>
<td>114A4318</td>
</tr>
</tbody>
</table>

**Program class storage: 114A4148**

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>114A4148</td>
</tr>
<tr>
<td>000020</td>
<td>114A4168</td>
</tr>
<tr>
<td>000040</td>
<td>114A4188</td>
</tr>
<tr>
<td>000060</td>
<td>114A41A8</td>
</tr>
<tr>
<td>000080</td>
<td>114A41C8</td>
</tr>
<tr>
<td>000100</td>
<td>114A4248</td>
</tr>
<tr>
<td>000120</td>
<td>114A4268</td>
</tr>
<tr>
<td>000140</td>
<td>114A4288</td>
</tr>
<tr>
<td>000160</td>
<td>114A4308</td>
</tr>
<tr>
<td>000180</td>
<td>114A4328</td>
</tr>
</tbody>
</table>

### Notes

No XML data was used in this program.

No FACTORY data was used in this program.

No DSA indexes were used in this program.

No LOCAL-STORAGE was used in this program.

No EXTERNAL data was used in this program.

No OBJECT instance data were used in this program.

No indexes were used in this program.

No variably-located areas were used in this program.

No XML data was used in this program.
Enclave-level data

The Enclave Control Blocks section of the dump, shown in Figure 86 on page 255, 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.

- 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.
Enclave Storage:

**GMAREA for file ESDS1DD in program COBDUMP2:**

- 00000000 00000000 00000000 00000000 00000000

**FIB for file ESDS1DD in program COBDUMP2:**

- 11202A34
- 00016A80 0001618C 00000000 000167FC 00000000 00000000 00000000 00000000

```
+000000 114A4018
+000020 114A4038
+000040 114A4058
+000060 114A4078
+000080 114A4098
+0000A0 114A40B8
+0000C0 114A40E8
+0000E0 114A4108
+000100 114A4128
+000120 114A4148
+000140 114A4168
+000160 114A4188
+000180 114A41A8
+0001A0 114A41C8
+0001C0 114A41E8
+000200 114A4208
+000220 114A4228
+000240 114A4248
+000260 114A4268
+000280 114A4288
+0002A0 114A42A8
+0002C0 114A42C8
+000300 114A4308
+000320 114A4328
+000340 114A4348
+000360 114A4368
+000400 114A4388
+000420 114A43A8
+000440 114A43C8
+000460 114A43E8
+000480 114A4408
+0004A0 114A4428
+0004C0 114A4448
+0004E0 114A4468
+000500 114A4488
+000520 114A44A8
+000540 114A44C8
+000560 114A44E8
+000580 114A4508
+0005A0 114A4528
+0005C0 114A4548
+0005E0 114A4568
+000600 114A4588
+000620 114A45A8
+000640 114A45C8
+000660 114A45E8
+000680 114A4608
+0006A0 114A4628
+0006C0 114A4648
+0006E0 114A4668
+000700 114A4688
+000720 114A46A8
+000740 114A46C8
+000760 114A46E8
+000780 114A4708
+0007A0 114A4728
+0007C0 114A4748
+0007E0 114A4768
+000800 114A4788
+000820 114A47A8
+000840 114A47C8
+000860 114A47E8
+000880 114A4808
+0008A0 114A4828
+0008C0 114A4848
+0008E0 114A4868
+000900 114A4888
+000920 114A48A8
+000940 114A48C8
+000960 114A48E8
+000980 114A4908
+0009A0 114A4928
+0009C0 114A4948
+0009E0 114A4968
+000A00 114A4988
+000A20 114A49A8
+000A40 114A49C8
+000A60 114A49E8
+000A80 114A4A08
+000AA0 114A4A28
+000AC0 114A4A48
+000AEC 114A4A88
+000B00 114A4B08
+000B20 114A4B28
+000B40 114A4B48
+000B60 114A4B68
+000B80 114A4B88
+000BA0 114A4BA8
+000BC0 114A4BB8
+000C00 114A4C08
+000C20 114A4C28
+000C40 114A4C48
+000C60 114A4C68
+000C80 114A4C88
+000CA0 114A4CA8
+000CC0 114A4CC8
+000CE0 114A4CE8
+000D00 114A4D08
+000D20 114A4D28
+000D40 114A4D48
+000D60 114A4D68
+000D80 114A4D88
+000DA0 114A4DA8
+000DC0 114A4DC8
+000E00 114A4E08
+000E20 114A4E28
+000E40 114A4E48
+000E60 114A4E68
+000E80 114A4E88
+000EA0 114A4EA8
+000EC0 114A4EC8
+000EE0 114A4EE8
+000F00 114A4F08
+000F20 114A4F28
+000F40 114A4F48
+000F60 114A4F68
+000F80 114A4F88
+000FA0 114A4FA8
+000FC0 114A4FC8
+000FE0 114A4FE8
```

**Enclave Control Blocks:**

- RUNCOM: 11408000
  - 000000 11408010
  - 000020 11408012
  - 000040 11408014
  - 000060 11408016
  - 000080 11408018
  - 000100 1140801A

**Initial (User) Heap:**

- 114A4018

**File Control Blocks:**

- FCB for file ESDS1DD in program COBDUMP2:
  - 11408000

- FIB for file ESDS1DD in program COBDUMP2:
  - 11202A34

**GNAME for file ESDS1DD in program COBDUMP2:**

- 000000 00000000

---

**Figure 86. Enclave-level data for COBOL programs**

---

Chapter 5. Debugging COBOL programs
Process-level data
The Process Control Block section of the dump, shown in Figure 87, displays COBOL process-level control blocks THDCOM, COBCOM, COBVEC, and ITBLK. 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 88 on page 257 illustrates the error of using a subscript value outside the range of an array. This program was compiled with LIST, TEST(STMT,SYM), 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, unless you specify CHECK(OFF) as a run-time option.

The program was run with TERMTHDACT(TRACE) to generate the traceback information shown in Figure 89 on page 258.
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 Figure 89 on page 258. 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 Run-Time Messages.

2. Statement 11 in the traceback section of the dump occurred in program COBOLX.
CEE3DMP V1 R9.0: Condition processing resulted in the unhandled condition. 02/14/07 2:59:00 PM

ASID: 0099  Job ID: J0005735  Job name: COBOLX  Step name: GO  UserID: BARBARA

CEEB3845I  CEEBDUMP Processing started.

Information for thread 8000000000000000

Traceback:

<table>
<thead>
<tr>
<th>DSA</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>CEEHDS</td>
<td>+000049D6</td>
<td>CEEPLPKA</td>
<td>CEEHDS</td>
<td>D1908 Call</td>
<td></td>
<td>01908</td>
</tr>
<tr>
<td>2</td>
<td>CEEHSSLG</td>
<td>+000003C2</td>
<td>CEEPLPKA</td>
<td>CEEHSSLG</td>
<td>D1908 Exception</td>
<td></td>
<td>01908</td>
</tr>
<tr>
<td>3</td>
<td>IGZCMGS</td>
<td>+000003C2</td>
<td>IGZCMGS</td>
<td>IGZCMGS</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>COBOLX</td>
<td>+000002BC</td>
<td>GO</td>
<td>COBOLX</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DSA - DSA Addr  E Addr  Pu Addr  Pu Offset  Comp Date  Compile Attributes
1 1147E880 11280238 11280238 +00004906 20061215 CEL
2 1147E880 112CF968 112CF968 +0000005C 20061215 CEL
3 1147E100 11455A0 11455A0 +000003C2 20061213 LIBRARY
4 1147E300 00008398 00008398 +000002BC 20070204 COBOL

Condition Information for Active Routines

Condition Information for CEEHSSLG (DSA address 1147E880)

- CIB Address: 1147F1A0
- Current Condition:
  - IGZ00065 The reference to table SLOT by verb number 01 on line 000011 addressed an area outside the region of the table.
- Location:
  - Program Unit: CEEHSSLG  Entry: CEEHSSLG  Statement: +0000005C
- Storage dump near condition, beginning at location: 112CF9AC

Parameters, Registers, and Variables for Active Routines:

CEEHDS (DSA address 1147E880):

- UPSTACK DSA
- Saved Registers:
  - GPR0..... 00008550 GPR1..... 1147E100 GPR2..... 1147E300 GPR3..... 00000010 GPR4..... 000083D0 GPR5..... 000080B0 GPR6..... 00000000 GPR7..... 00000000 GPR8..... 0000A398 GPR9..... 00000000 GPR10.... 000080B0 GPR11.... 000080B0 GPR12.... 00000000 GPR13.... 1147E880 GPR14.... 1147E880 GPR15.... 1147E880
- GPREG STORAGE:
  - Storage around GPR0 (00008550)

COBOLX (DSA address 1147E880):

- UPSTACK DSA
- Saved Registers:
  - GPR0..... 1147E100 GPR1..... 00000100 GPR2..... 00000036 GPR3..... 0000197F GPR4..... 11208778 GPR5..... 11208778 GPR6..... 00000000 GPR7..... 00000000 GPR8..... 00001398 GPR9..... 00001398 GPR10.... 00000040 GPR11.... 00000040 GPR12.... 00000040 GPR13.... 1147E300 GPR14.... 00008656 GPR15.... 11455A0
- GPREG STORAGE:
  - Storage around GPR0 (1147E100)

Figure 89. Sections of Language Environment dump for COBOLX (Part 1 of 2)
3. Find the statement on line 11 in the listing for program COBOLX, shown in Figure 90. 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 Figure 89 on page 258. 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.

**Calling a nonexistent subroutine**

Figure 91 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 91 shows the program.

```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 91. COBOL example of calling a nonexistent subroutine**

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 92**. The message is **CEE3501S The module UNKNOWN was not found.** For more information about this message, see **z/OS Language Environment Run-Time Messages**.

2. Note the sequence of calls in the **Traceback** section of the dump. COBOLY 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 COBOLY was attempting to make a dynamic call. The call statement in COBOLY is located at offset +0000036E.

3. Use the offset of X'36E' from the COBOL listing, shown in **Figure 93 on page 261**, to locate the statement that caused the exception in the COBOLY program. At offset X'36E' is an instruction for statement 8. Statement 8 is a call with the identifier **SUBNAME** specified.

---

**Figure 92. Sections of Language Environment dump for COBOLY**

<table>
<thead>
<tr>
<th>DSA</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>CEEHP</td>
<td>+00004030</td>
<td>CEEPLPKA</td>
<td>CEEHSP</td>
<td>D1908</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CEEHSGLT</td>
<td>+0000005C</td>
<td>CEEPLPKA</td>
<td>CEEHSGLT</td>
<td>D1908</td>
<td>Exception</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IGZCLDL</td>
<td>+0000012A</td>
<td>IGZCPAC</td>
<td>IGZCLDL</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IGZCFCC</td>
<td>+000003EE</td>
<td>IGZCPAC</td>
<td>IGZCFCC</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>COBOLY</td>
<td>+0000036E</td>
<td>8</td>
<td>GO</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Condition Information for Active Routines**

**Condition Information for CEEHSGLT (DSA address 1147EE58)**

CIB Address: 1147EE58

Current Condition: **CEE0198S The termination of a thread was signaled due to an unhandled condition.**

**Original Condition:** **CEE3501S The module UNKNOWN was not found.**

Location:

<table>
<thead>
<tr>
<th>Program Unit</th>
<th>CEEHSGLT Entry: CEEHSGLT Statement: Offset: +0000005C</th>
</tr>
</thead>
</table>

Storage dump near condition, beginning at location: 112CF9AC

+000000 112CF9AC F010020B D0B0100D 5B00C28B 5B0F001C 05EF020B D098010B 41A0000B 50A000B0C |0.K.......B.....K....q......q&...|...
Figure 93. COBOL Listing for COBOLY (Part 1 of 2)
4. Find the value of the local variables in the Parameters, Registers, and Variables for Active Routines section of the dump, shown in Figure 94. Notice that the value of SUBNAME with usage DISP '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 93. COBOL Listing for COBOLY (Part 2 of 2)

Figure 94. Parameters, registers, and variables for active routines section of dump for COBOLY

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 TERMTHDCT(TRACE) run-time option. Figure 95 on page 263 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 Figure 96 on page 264. 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 Run-Time 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.
Condition Information for enclave COBOLZ1

Information for thread 8000000000000000

Traceback:

<table>
<thead>
<tr>
<th>DSA</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>CEEHDSP</td>
<td>+000049D6</td>
<td>CEEPLPKA</td>
<td>CEEHDSP</td>
<td></td>
<td>$190B</td>
<td>Call</td>
</tr>
<tr>
<td>2</td>
<td>ASSEMZ3</td>
<td>+00000064</td>
<td>ASSEMZ3</td>
<td>ASSEMZ3</td>
<td></td>
<td></td>
<td>Exception</td>
</tr>
<tr>
<td>3</td>
<td>IGZCFCC</td>
<td>+000002CA</td>
<td>IGZCPAC</td>
<td>IGZCFCC</td>
<td></td>
<td></td>
<td>Call</td>
</tr>
<tr>
<td>4</td>
<td>COBOLZ2</td>
<td>+000002A4</td>
<td>COBOLZ2</td>
<td>COBOLZ2</td>
<td></td>
<td></td>
<td>Call</td>
</tr>
<tr>
<td>5</td>
<td>IGZCFCC</td>
<td>+000002CA</td>
<td>IGZCPAC</td>
<td>IGZCFCC</td>
<td></td>
<td></td>
<td>Call</td>
</tr>
<tr>
<td>6</td>
<td>COBOLZ1</td>
<td>+000002B8</td>
<td>GO</td>
<td></td>
<td></td>
<td></td>
<td>Call</td>
</tr>
</tbody>
</table>

DSA DSA Addr E Addr PU Addr PU Offset Comp Date Compile Attributes
1 1147E7D0 112B0238 112B0238 +000049D6 20061215 CEL
2 1147E750 1120E4A8 1120E4A8 +00000064 20070214 ASM
3 1147E570 1140A5F8 1140A5F8 +000002CA 20061213 LIBRARY
4 1147E3C0 0001F320 0001F320 +000002A4 20070214 COBOL
5 1147E1E0 1140A5F8 1140A5F8 +000002CA 20061213 LIBRARY
6 1147E930 000083D0 000083D0 +000002B8 20070214 COBOL

Condition Information for Active Routines

Condition Information for ASSEMZ3 (DSA address 1147E750)
CIB Address: 114770F0
Current Condition:

CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).
Location:
Program Unit: ASSEMZ3
Entry: ASSEMZ3
Statement: Offset: +00000064
Machine State:
ILC..... 0004 Interruption Code..... 0009
PSW..... 078D0000 9120E4B0
GPR0..... 00000000_1147E7D0 GPR1..... 00000000_1147E4B8 GPR2..... 00000000_000197FC
GPR3..... 00000000_000212E8 GPR4..... 00000000_00000000 GPR5..... 00000000_000191BC
GPR6..... 00000000_00000000 GPR7..... 00000000_1147E4B8 GPR8..... 00000000_000213A8
GPR9..... 00000000_000211B0 GPR10.... 00000000_0001F428 GPR11.... 00000000_9120E448
GPR12.... 00000000_1147E570 GPR13.... 00000000_1147E4B8 GPR14.... 8001F5C6 GPR15.... 9140A5F8

Storage dump near condition, beginning at location: 1120E49C
+000000 1120E49C 10184150 092C1B44 58610000 41660000 5D460000 58F0B0C8 5800B0C8 58DD0004 |...&...../......)....0.H...H....|
GPREG STORAGE:
Storage around GPR0 (1147E7D0)
-0020 1147E550 1147E30C 1147E44C 1120B658 00000000 1147E470 00000000 00000110 1147E958 |..T...U<..........U.............|
+0000 1147E750 00102401 1147E3C0 1147E750 9140A8C4 9120E448 1147E750 1147E4B8 1147AE54 |......T...X&j yDj.U...X&..U.....|
+0020 1147E590 000212E8 1147E44C 000212E8 1147B028 1147E4B8 000211B0 1147A100 1147A100 |...Y..U....Y......U..........|

Parameters, Registers, and Variables for Active Routines:

COBOLZ2 (DSA address 1147E3C0):
UPSTACK DSA
Saved Registers:
GPR0..... 1147E570 GPR1..... 1147E4C0 GPR2..... 0001977C GPR3..... 000212E8
GPR4..... 1147E4B8 GPR5..... 000191BC GPR6..... 1147AFE0 GPR7..... 00001000
GPR8..... 000213A8 GPR9..... 000211B0 GPR10.... 0001F428 GPR11.... 0001F54C
GPR12.... 0001F41C GPR13.... 1147E3C0 GPR14.... 8001F5C6 GPR15.... 9140A5F8

GPREG STORAGE:
Storage around GPR0 (1147E3C0)
-0020 1147E550 1147E30C 1147E44C 1120B658 00000000 1147E470 00000000 00000110 1147E958 |..T...U<..........U.............|
+0000 1147E570 00102401 1147E3C0 1147E750 9140A8C4 9120E448 1147E750 1147E4B8 1147AE54 |......T...X&j yDj.U...X&..U.....|
+0020 1147E590 000212E8 1147E44C 000212E8 1147B028 1147E4B8 000211B0 1147A100 1147A100 |...Y..U....Y......U..........|

Local Variables:
6 77 DV-VAL 9999 COMP 00000
8 77 D-VAL 9999 COMP 00000

Figure 96. Sections of Language Environment dump for program COBOLZ1 (Part 1 of 2)
3. Locate statement 11 in the COBOL listing for the COBOLZ2 program, shown in Figure 97. This is a call to the assembler routine ASSEMZ3.

4. Check offset +64 in the listing for the assembler routine ASSEMZ3, shown in Figure 98 on page 266.

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.

**Note:** To translate assembler instructions, see z/Architecture Principles of Operation.
5. Check local variables for COBOLZ2 in the Local Variables section of the dump shown in Figure 99. 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 100 on page 267, indicating that the value did pass from COBOLZ1 to COBOLZ2.
In the Local Variables section of the dump for program COBOLZ1, shown in Figure 101, 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.

Figure 101. Variables section of Language Environment dump for COBOLZ1
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 run-time 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 run-time errors

Fortran has several features that help you find run-time errors. Fortran run-time messages are discussed in "z/OS Language Environment Run-Time 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) run-time 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 271.

Operator messages are displayed when your program issues a PAUSE or STOP 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 run-time 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 run-time 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 40 lists the contents of the various compiler-generated listings that you might find helpful when you use information in dumps to debug Fortran programs.

Table 40. Compiler-generated Fortran listings and their contents

<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.

```fortran
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.

```fortran
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.

```fortran
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**

```fortran
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.

```fortran
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

```fortran
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 run-time 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 102 on page 275 shows a sample program calling SDUMP and Figure 103 on page 276 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 102. Example program that calls SDUMP (Part 1 of 2)
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 104 on page 277 shows an example of a Fortran dump; Table 41 on page 277 provides additional information.
about each section within the dump.

Table 41 describes the sections shown in the sample dump in Figure 104.

Table 41. 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 41. 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 105 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 NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NODIM NORENT SDUMP(ISN)
   NOSYM NOVECTOR IL(DIM) NOTEST SC(+*) NODC NOEC NOEMODE NOICA NODIRECTIVE NODBCS NOSAA NOPARALLEL NODYNAMIC NOSYM
   NOORDER NOPC
   OPT(0) LANGLVL(77) NOFIPS FLAG(I) AUTODBL(DONE) PTRSIZE(8) LINECOUNT(60) CHARLEN(500) NAME(MAIN)
1    PROGRAM CALLNON
2 INTEGER*4 ARRAY_END
C
3 CALL SUBNAM
4 STOP
5 END
```

Figure 105. Example of calling a nonexistent routine

Figure 106 on page 279 shows sections of the dump generated by a call to SDUMP.
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 Run-Time 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.

2. Locate statement 3 in the routine shown in Figure 105 on page 278. 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.
**Divide-by-zero error**

Figure 107 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.

```fortran
PROGRAM DIVZERO
  INTEGER*4 ANY_NUMBER
  INTEGER*4 ANY_ARRAY(3)
  PRINT *,'EXAMPLE STARTING'
  ANY_NUMBER = 0
  DO I = 1, 3
    ANY_ARRAY(I) = I
  END DO
  CALL DIVZEROSUB(ANY_NUMBER, ANY_ARRAY)
  PRINT *,'EXAMPLE ENDING'
STOP
END
```

```fortran
SUBROUTINE DIVZEROSUB(DIVISOR, DIVIDEND)
  INTEGER*4 DIVISOR
  INTEGER*4 DIVIDEND(3)
  PRINT *,'IN SUBROUTINE DIVZEROSUB'
  DIVIDEND(1) = DIVIDEND(3) / DIVISOR
  PRINT *,'END OF SUBROUTINE DIVZEROSUB'
RETURN
END
```

Figure 107. Fortran routine with a divide-by-zero error

Figure 108 on page 281 shows the Language Environment dump for routine DIVZERO.
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 108. The system detected a fixed-point divide exception. See z/OS Language Environment Run-Time 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.
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 108 on page 281. 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 108 on page 281. 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 run-time messages, which begin with the prefix IBM. For a list of these messages, see z/OS Language Environment Run-Time 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 run-time routine malfunction
- System malfunction
- Unidentified routine malfunction
- Overlayed 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 run-time routine malfunction
If you are certain that the malfunction is caused by a compiler or run-time routine
error, you can either open a PMR or submit an APAR for the error. See PL/I for
MVS & VM Diagnosis Guide for more information about handling compiler and
run-time routine malfunctions. 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 run-time 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;
```
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>,* OPTIONAL) OPTIONS(ASM); CALL CEEDATE(x,y,z,<em>); /</em> valid <em>/ CALL CEEDATE(x,y,z,fc); /</em> 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 42 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 PL/I for MVS & VM Programming Guide.

<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 42. 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 109 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 109. PL/I for MVS & VM routine compiled with LIST and MAP

Figure 110 on page 288 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 110. Compiler-generated listings from example PL/I for MVS & VM routine (Part 1 of 2)
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 43 lists some typical comments you might find in a static storage listing.

<table>
<thead>
<tr>
<th>Table 43. Typical comments in a PL/I for MVS &amp; VM static storage listing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comment</strong></td>
</tr>
<tr>
<td>A..xxx</td>
</tr>
<tr>
<td>COMPILER LABEL CL..n</td>
</tr>
<tr>
<td>CONDITION CSECT</td>
</tr>
<tr>
<td>CONSTANT</td>
</tr>
<tr>
<td>CSECT FOR EXTERNAL VARIABLE</td>
</tr>
<tr>
<td>D..xxx</td>
</tr>
<tr>
<td>DED..xxx</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
</tr>
<tr>
<td>ENVB</td>
</tr>
<tr>
<td>FECB..xxx</td>
</tr>
<tr>
<td>DCLCB</td>
</tr>
<tr>
<td>FED..xxx</td>
</tr>
</tbody>
</table>

Figure 110. Compiler-generated listings from example PL/I for MVS & VM routine (Part 2 of 2)
Table 43. 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 ADCON</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 44 summarizes the comment used in the listing.

Table 44. 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>Identifies 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>Identifies 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 45 on page 292) 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 45. 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..n</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 35](#).

### 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. An ddbname 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 111](#) 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 112.

<table>
<thead>
<tr>
<th>Task Attached by</th>
<th>Thread ID</th>
<th>TCA Addr</th>
<th>EV Addr</th>
<th>TV Addr</th>
<th>Absolute Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBTSK2</td>
<td>SUBTSK1</td>
<td>1171C160000000003</td>
<td>11489BD8</td>
<td>11200AA0</td>
<td>00035290</td>
</tr>
<tr>
<td>SUBTSK1</td>
<td>SUBTASK</td>
<td>1171943000000002</td>
<td>11480BD8</td>
<td>11200AE0</td>
<td>000266F8</td>
</tr>
<tr>
<td>SUBTASK TASKING</td>
<td>SUBTASKE</td>
<td>1171761000000001</td>
<td>11440D48</td>
<td>11200B20</td>
<td>00025D70</td>
</tr>
<tr>
<td>TASKING</td>
<td>TASKING</td>
<td>1170C15000000000</td>
<td>1120F9C0</td>
<td>000255D0</td>
<td>0002523C</td>
</tr>
</tbody>
</table>

Information for enclave TASKING

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>TCA Addr</th>
<th>EV Addr</th>
<th>TV Addr</th>
<th>Absolute Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1171C16000000003</td>
<td>11489BD8</td>
<td>11200AA0</td>
<td>00035290</td>
<td></td>
</tr>
</tbody>
</table>

Traceback:

<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>11443530</td>
<td>1143A850</td>
<td>1143A850</td>
<td>+00000146</td>
<td>20061214</td>
<td>LIBRARY POSIX</td>
</tr>
</tbody>
</table>

Information for thread 1171943000000002

<table>
<thead>
<tr>
<th>Task Attached by</th>
<th>Thread ID</th>
<th>TCA Addr</th>
<th>EV Addr</th>
<th>TV Addr</th>
<th>Absolute Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBTSK2</td>
<td>SUBTSK1</td>
<td>1171C160000000003</td>
<td>11489BD8</td>
<td>11200AA0</td>
<td>00035290</td>
</tr>
<tr>
<td>SUBTSK1</td>
<td>SUBTASK</td>
<td>1171943000000002</td>
<td>11480BD8</td>
<td>11200AE0</td>
<td>000266F8</td>
</tr>
<tr>
<td>SUBTASK TASKING</td>
<td>SUBTASKE</td>
<td>1171761000000001</td>
<td>11440D48</td>
<td>11200B20</td>
<td>00025D70</td>
</tr>
<tr>
<td>TASKING</td>
<td>TASKING</td>
<td>1170C15000000000</td>
<td>1120F9C0</td>
<td>000255D0</td>
<td>0002523C</td>
</tr>
</tbody>
</table>

Information for thread 1171C16000000003

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>TCA Addr</th>
<th>EV Addr</th>
<th>TV Addr</th>
<th>Absolute Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1171C16000000003</td>
<td>11489BD8</td>
<td>11200AA0</td>
<td>00035290</td>
<td></td>
</tr>
</tbody>
</table>

Traceback:

<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>11443530</td>
<td>1143A850</td>
<td>1143A850</td>
<td>+00000146</td>
<td>20061214</td>
<td>LIBRARY POSIX</td>
</tr>
</tbody>
</table>

Information for thread 11719430000000002

<table>
<thead>
<tr>
<th>Task Attached by</th>
<th>Thread ID</th>
<th>TCA Addr</th>
<th>EV Addr</th>
<th>TV Addr</th>
<th>Absolute Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBTSK2</td>
<td>SUBTSK1</td>
<td>1171C160000000003</td>
<td>11489BD8</td>
<td>11200AA0</td>
<td>00035290</td>
</tr>
<tr>
<td>SUBTSK1</td>
<td>SUBTASK</td>
<td>1171943000000002</td>
<td>11480BD8</td>
<td>11200AE0</td>
<td>000266F8</td>
</tr>
<tr>
<td>SUBTASK TASKING</td>
<td>SUBTASKE</td>
<td>1171761000000001</td>
<td>11440D48</td>
<td>11200B20</td>
<td>00025D70</td>
</tr>
<tr>
<td>TASKING</td>
<td>TASKING</td>
<td>1170C15000000000</td>
<td>1120F9C0</td>
<td>000255D0</td>
<td>0002523C</td>
</tr>
</tbody>
</table>

Information for thread 1171C16000000003

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>TCA Addr</th>
<th>EV Addr</th>
<th>TV Addr</th>
<th>Absolute Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1171C16000000003</td>
<td>11489BD8</td>
<td>11200AA0</td>
<td>00035290</td>
<td></td>
</tr>
</tbody>
</table>

Traceback:

<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>11443530</td>
<td>1143A850</td>
<td>1143A850</td>
<td>+00000146</td>
<td>20061214</td>
<td>LIBRARY POSIX</td>
</tr>
</tbody>
</table>

Information for thread 11719430000000002

<table>
<thead>
<tr>
<th>Task Attached by</th>
<th>Thread ID</th>
<th>TCA Addr</th>
<th>EV Addr</th>
<th>TV Addr</th>
<th>Absolute Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBTSK2</td>
<td>SUBTSK1</td>
<td>1171C160000000003</td>
<td>11489BD8</td>
<td>11200AA0</td>
<td>00035290</td>
</tr>
<tr>
<td>SUBTSK1</td>
<td>SUBTASK</td>
<td>1171943000000002</td>
<td>11480BD8</td>
<td>11200AE0</td>
<td>000266F8</td>
</tr>
<tr>
<td>SUBTASK TASKING</td>
<td>SUBTASKE</td>
<td>1171761000000001</td>
<td>11440D48</td>
<td>11200B20</td>
<td>00025D70</td>
</tr>
<tr>
<td>TASKING</td>
<td>TASKING</td>
<td>1170C15000000000</td>
<td>1120F9C0</td>
<td>000255D0</td>
<td>0002523C</td>
</tr>
</tbody>
</table>

Information for thread 1171C16000000003

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>TCA Addr</th>
<th>EV Addr</th>
<th>TV Addr</th>
<th>Absolute Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1171C16000000003</td>
<td>11489BD8</td>
<td>11200AA0</td>
<td>00035290</td>
<td></td>
</tr>
</tbody>
</table>
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 113 on page 298 shows this section of the dump.

Figure 112. Task traceback section (Part 2 of 2)
Figure 113. Control blocks for active routines section of the dump (Part 1 of 3)
Figure 113. Control blocks for active routines section of the dump (Part 2 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'C8' 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
```
The area starts at offset X’F8’ 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 *PL/I for MVS & VM Programming Guide*.

**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 114 on page 302 includes information about PL/I for MVS & VM fields of the CAA and other control block information.
**Figure 114. Control blocks associated with the thread section of the dump (Part 1 of 2)**

### Control Blocks Associated with the Thread:

**CA: 2090E9C0**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2090E9C0</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>2090E9D0</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>2090E9E0</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>2090E9F0</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>2090F000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>2090F010</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>2090F020</td>
<td>000000000000000</td>
<td></td>
</tr>
</tbody>
</table>

### Enclave Control Blocks:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
</tbody>
</table>

### PL/I TCA Apppendage:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
</tbody>
</table>

### DUMMY DSA: 2090F360

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>000000000000000</td>
<td></td>
</tr>
</tbody>
</table>

---

The figure provides a detailed view of the control blocks associated with the thread section of the dump.
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 163.

--- (100) NameOfCallingTask NameOfCalledTask OffsetOfCallStmt
UserAgrPtr CalledTaskPtr TaskVarPtr EventVarPtr PriorityPtr CallingR2-R5 CallingR12-R14

--- (101) NameOfReturnTask ReturnerR2-R5 ReturnerR12-R14

--- (102) NameOfReturnTask

**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 115 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.

---

**Figure 115. Example of moving a value outside an array range**
Figure 116 shows sections of the dump generated by a call to PLIDUMP.

Figure 116. Sections of the Language Environment dump (Part 1 of 2)
Figure 116. 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 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 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 Language Environment Run-Time Messages.

3. Locate statement 9 in the routine in Figure 115 on page 304. The instruction is

4. Statement 10 begins the DO-loop instruction Do 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 115 on page 304. 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 116 on page 305. The block located at this offset contains the value that exceeded the array range, X'B' or 11.

### Calling a nonexistent subroutine

Figure 117 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) run-time option to generate a traceback.

Figure 117. Example of calling a nonexistent subroutine

Figure 118 on page 305 shows the traceback and condition information from the dump.
PLIDUMP was called from statement number 5 at offset +000000D6 from ERR On-unit with entry address 20900D2C

Information for enclave EXAMPLE1

Information for thread 80000000

Traceback:

<table>
<thead>
<tr>
<th>DSA Entry</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CEEKKMRA</td>
<td>+0000081C</td>
<td>CEEPLPKA</td>
<td>CEEKKMRA</td>
<td>D1908 Call</td>
</tr>
<tr>
<td>2</td>
<td>IBMRKDCM</td>
<td>+0000091C</td>
<td>IBMRKDCM</td>
<td>IBMRKDCM</td>
<td>Call</td>
</tr>
<tr>
<td>3</td>
<td>ERR On-unit</td>
<td>+000000D6</td>
<td>EXAMPLE1</td>
<td>EXAMPLE1</td>
<td>Call</td>
</tr>
<tr>
<td>4</td>
<td>IBMRKDCM</td>
<td>+0000091C</td>
<td>IBMRKDCM</td>
<td>IBMRKDCM</td>
<td>Call</td>
</tr>
<tr>
<td>5</td>
<td>CEEMKDCM</td>
<td>+0000013A</td>
<td>CEEPLPKA</td>
<td>CEEMKDCM</td>
<td>Call</td>
</tr>
<tr>
<td>6</td>
<td>EXAMPLE1</td>
<td>-20900D2C</td>
<td>EXAMPLE1</td>
<td>EXAMPLE1</td>
<td>Exception</td>
</tr>
<tr>
<td>7</td>
<td>IBMRKDCM</td>
<td>+0000051E</td>
<td>IBMRKDCM</td>
<td>IBMRKDCM</td>
<td>Call</td>
</tr>
<tr>
<td>8</td>
<td>CEEKKMRA</td>
<td>+0000081C</td>
<td>CEEPLPKA</td>
<td>CEEKKMRA</td>
<td>D1908 Call</td>
</tr>
<tr>
<td>9</td>
<td>EXAMPLE1</td>
<td>-20900D2C</td>
<td>EXAMPLE1</td>
<td>EXAMPLE1</td>
<td>Exception</td>
</tr>
<tr>
<td>10</td>
<td>CEEBEXT</td>
<td>+0000013A</td>
<td>CEEPLPKA</td>
<td>CEEBEXT</td>
<td>Call</td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Condition Information for EXAMPLE1 (DSA address 20900D2C)

CIB Address: 20B42D10

Current Condition:

CEE3201S The system detected an operation exception (System Completion Code=0C1).

Location:

Program Unit: EXAMPLE1 Entry: EXAMPLE1 Statement: Offset: -20900D2C

Possible Bad Branch: Statement: 7 Offset: +000000C0

Machine State:

ILC..... 0002 Interruption Code..... 0001
PSW..... 078D0E00 80000002
GPR0..... 00000000_20B423F0 GPR1..... 00000000_00000000 GPR2..... 00000000_A0900D4 GPR3..... 00000000_20900D2C GPR4..... 00000000_00000000 GPR5..... 00000000_00000000 GPR6..... 00000000_20B423E8 GPR7..... 00000000_00000000

Storage dump near condition, beginning at location: 00000000

Inaccessible storage.

Figure 118. 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 Run-Time 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 117 on page 307). 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 119 on page 310 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 120 shows this output.

The routine in Figure 119 was compiled with the LIST compiler option, which generated the object code listing shown in Figure 121 on page 311.
Figure 121. Object code listing from example PL/I for MVS & VM routine

Figure 122 on page 312 shows the Language Environment dump for routine SAMPLE.
PLIDUMP was called from statement number 4 at offset +000000D6 from ERR ON-unit with entry address 2090022C

Information for enclave SAMPLE

Traceback:

1. CEEKKMRA +0000081C CEEPLPKA CEEKKMRA D1908 Call
2. IBMXKDM +000000C2 IBMREV10 IBMXKDM Call
3. ERR ON-unit+000000D6 4 SAMPLE SAMPLE Call
4. IBMXERPL +0000065A IBMRLIB1 IBMXERPL Call
5. CEEEV010 +0000013A IBMREV10 CEEEV010 Call
6. CEENDSP +00000170 CEEPLPKA CEENDSP D1908 Call
7. SUB1 +000000EE 15 SIMPLE SAMPLE Call
8. SIMPLE +00000154 11 SIMPLE SAMPLE Call
9. IBMRPMIA +0000051E IBMRLIB1 IBMRPMIA Call
10. CEEEV010 +00000310 IBMREV10 CEEEV010 Call
11. CEEBBEXT +000001A6 CEEPLPKA CEEBBEXT D1908 Call

Condition Information for Active Routines

Condition Information for SAMPLE (DSA address 20842460)

CIB Address: 20842E08

Current Condition:

IBM0281S A prior condition was promoted to the ERROR condition.

Original Condition:

CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).

Location:

Program Unit: SIMPLE Entry: SUB1 Statement: 15 Offset: +000000EE

Machine State:

ILC...... 0004 Interruption Code..... 0009
PSW...... 078D2E00 A0900452

Figure 122. Language Environment dump from example PL/I for MVS & VM routine (Part 1 of 3)
Figure 122. 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 Run-Time 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 121 on page 311. 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 run-time messages, which begin with the prefix IBM. For a list of these messages, see z/OS Language Environment Run-Time 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 run-time 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.

```plaintext
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 run-time routine malfunction**

If you are certain that the malfunction is caused by a compiler or run-time 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 run-time 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:

```plaintext
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
str(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); 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);</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 46 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>Table 46. Compiler-generated PL/I listings and their contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Source program</td>
</tr>
<tr>
<td>Cross reference</td>
</tr>
<tr>
<td>Aggregate table</td>
</tr>
<tr>
<td>Variable map</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Object code</td>
</tr>
</tbody>
</table>
Table 46. 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 123 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 123. Enterprise PL/I routine compiled with LIST, MAP, and SOURCE

Figure 124 on page 322 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 124. Compiler-generated listings from example Enterprise PL/I routine (Part 1 of 4)
### Figure 124. Compiler-generated listings from example Enterprise PL/I routine (Part 2 of 4)
General purpose registers used: 1111111110001111
Floating point registers used: 1111111100000000
Size of dynamic storage: 200
Size of executable code: 350
CSECT Offset: 72 : 0x48

Constant Area

OFFSET OBJECT CODE LINE# FILE# PSEUDO ASSEMBLY LISTING

PPA1: Entry Point Constants
000000 1CCEA166 =F'483303782' Flags
000004 00001CB =A(PPA2-EXAMPLE)
000008 00000000 =F'0' No PPA3
00000C 00000000 =F'0' No EPD
000010 FFEEEE00 =F'-2097152' Register save mask
000014 00000000 =F'0' Member flags
000018 00000000 =AL1(144)
000019 00000000 =AL3(O) callee's DSA use/8
00001C 00000000 =F'0' Flags
00001E 00000000 =F'0' State variable location
000020 00000000 =F'0' Offset/2 to CDL
000024 00000000 =F'1342177455' CDL function length/2
000028 00000000 =F'942145536' CDL prolog
00002C 00000000 =F'942145536' CDL epilog
000030 00000000 =F'0' CDL end
000034 00000000 =F'0' CDL end
000038 00000000 =F'0' CDL end

PPA1 End

PPA2: Compile Unit Block
000000 0B00 3203 =F'184562179' Flags
000004 FFFFFDFO =A(CEESTART-PPA2)
000008 0000 0000 =F'0' No PPA4
00000C 0000 0000 =F'0' No primary
000010 0000 0000 =F'0' Flags
000014 0200 0000 =F'33554432' Flags

PPA2 End

EXTERNAL SYMBOL DICTIONARY

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>EXAMPLE2</td>
<td>SD</td>
<td>2</td>
<td>000000</td>
<td>00005C</td>
</tr>
<tr>
<td>EXAMPLE3</td>
<td>SD</td>
<td>3</td>
<td>000000</td>
<td>000004</td>
</tr>
<tr>
<td>EXAMPLE4</td>
<td>LD</td>
<td>0</td>
<td>000000</td>
<td>000001</td>
</tr>
<tr>
<td>IBMQFRG</td>
<td>ER</td>
<td>6</td>
<td>000000</td>
<td>000000</td>
</tr>
<tr>
<td>IBMQFSH</td>
<td>ER</td>
<td>8</td>
<td>000000</td>
<td>000000</td>
</tr>
<tr>
<td>CEEMAIN</td>
<td>SD</td>
<td>10</td>
<td>000000</td>
<td>000000</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>ER</td>
<td>12</td>
<td>000000</td>
<td>000000</td>
</tr>
</tbody>
</table>

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>EXAMPLE3</td>
<td>EXAMPLE3</td>
</tr>
<tr>
<td>EXAMPLE4</td>
<td>EXAMPLE4</td>
</tr>
<tr>
<td>IBMQFRG</td>
<td>IBMQFRG</td>
</tr>
<tr>
<td>IBMQFSH</td>
<td>IBMQFSH</td>
</tr>
<tr>
<td>CEESG011</td>
<td>CEESG011</td>
</tr>
<tr>
<td>CEESG012</td>
<td>CEESG012</td>
</tr>
<tr>
<td>CEESTART</td>
<td>CEESTART</td>
</tr>
<tr>
<td>CEEMAIN</td>
<td>CEEMAIN</td>
</tr>
<tr>
<td>IBMQINPL</td>
<td>IBMQINPL</td>
</tr>
</tbody>
</table>

Figure 124. 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).

### Figure 124. Compiler-generated listings from example Enterprise PL/I routine (Part 4 of 4)
• 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 35.

**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 125 on page 330, for condition information about your routine and information about the statement number and address where the exception occurred.
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 126 on page 332] 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 0C8   L 6, P
58 E0 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 127 on page 334](#) includes information
about Enterprise PL/I fields of the CAA and other control block information.
### Control Blocks Associated with the Thread:

**Enterprise PL/I**

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>Flags... member... BCK... FMC... R14... R12... R11... reserved... MODE... reserved...</td>
</tr>
<tr>
<td>+000010</td>
<td>R15... 750000008... R1... 11212768... R2... 91215770... 9120125A... 91213980... 00000000...</td>
</tr>
<tr>
<td>+000020</td>
<td>R4... 00000000... R6... 91215770... 92000000...</td>
</tr>
<tr>
<td>+000038</td>
<td>R9... 08FF7F0... R10... 0000000... R11... 9120125A... R12... 91213980... reserved... 00000000...</td>
</tr>
<tr>
<td>+000040</td>
<td>NAB... 11A3B030... 11A3B090... 11A3B030... reserved... 0000000...</td>
</tr>
<tr>
<td>+000064</td>
<td>reserved... reserved... reserved... 00000000... reserved... reserved... 00000000...</td>
</tr>
</tbody>
</table>

**PL/I TCA Appendage:**

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>11A5B04E... 11A5B047... 11A5B046... 11A5B045... 11A5B044... 11A5B043... 11A5B042... 11A5B041... 11A5B040...</td>
</tr>
<tr>
<td>+000020</td>
<td>11A5B038... 11A5B037... 11A5B036... 11A5B035... 11A5B034... 11A5B033... 11A5B032... 11A5B031... 11A5B030...</td>
</tr>
</tbody>
</table>

**PL/I OCA:**

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>11A5A2C0... 11A5A2C1... 11A5A2C2... 11A5A2C3... 11A5A2C4... 11A5A2C5... 11A5A2C6... 11A5A2C7... 11A5A2C8...</td>
</tr>
<tr>
<td>+000020</td>
<td>11A5A2CA... 11A5A2CB... 11A5A2CC... 11A5A2CD... 11A5A2CE... 11A5A2CF... 11A5A2D0... 11A5A2D1... 11A5A2D2...</td>
</tr>
</tbody>
</table>

**PL/I OCA:**

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>11A5A2D8... 11A5A2D9... 11A5A2DA... 11A5A2DB... 11A5A2DC... 11A5A2DD... 11A5A2DE... 11A5A2DF... 11A5A2E0...</td>
</tr>
</tbody>
</table>

**Enclave Control Blocks:**

**EDB:** 11212648

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>11212648... 11212649... 11212650... 11212651... 11212652... 11212653... 11212654... 11212655... 11212656...</td>
</tr>
</tbody>
</table>

**MEML:** 11213870

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>11213870... 11213871... 11213872... 11213873... 11213874... 11213875... 11213876... 11213877... 11213878...</td>
</tr>
</tbody>
</table>

---

Figure 127. Control blocks associated with the thread section of the dump (Enterprise PL/I) (Part 1 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

Figure 127. Control blocks associated with the thread section of the dump (Enterprise PL/I) (Part 2 of 2)
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 163.

```
––>(100) NameOfCallingTask NameOfCalledTask OffsetOfCallStmt
       UserAgPtr 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 128 on page 337 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 TERMTHDCT(TRACE) option to generate a traceback for the condition.
Figure 129 on page 338 shows sections of the dump generated by a call to PLIDUMP.
PLIDUMP was called from statement number 9 at offset +000000D2 from _ON_Begin_7_Blk_2 with entry address 11400240

Information for enclave EXAMPLE

Information for thread 8000000000000000

Traceback:

<table>
<thead>
<tr>
<th>DSA</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>IBMPDUMP</td>
<td>+000002AE</td>
<td>IBMPDUMP</td>
<td>11A3DE50</td>
<td>114A4E38</td>
<td>11A3DEE8</td>
<td>Call</td>
</tr>
<tr>
<td>2</td>
<td>_ON_Begin_7_Blk_2</td>
<td>+000000D2</td>
<td>EXAMPLE</td>
<td>11A3DD70</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>3</td>
<td>IBMPEONR</td>
<td>+000002A2</td>
<td>IBMPDUMP</td>
<td>11A3DEE8</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>4</td>
<td>IBMPEDOP</td>
<td>+0000040C</td>
<td>IBMPDUMP</td>
<td>11A3DEE8</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>5</td>
<td>CEEEV011</td>
<td>+00000170</td>
<td>CEEPLPKA</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>6</td>
<td>CEEHDSP</td>
<td>+000017D0</td>
<td>CEEPLPKA</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>7</td>
<td>IBMBERRI</td>
<td>+0000000A</td>
<td>IBMPDUMP</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>8</td>
<td>ERR.Raise_COND</td>
<td>+00000090</td>
<td>IBMPDUMP</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>9</td>
<td>IBMPERSU</td>
<td>+00000202</td>
<td>IBMPDUMP</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>10</td>
<td>_Begin_12_Blk_3</td>
<td>+00000100</td>
<td>EXAMPLE</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>11</td>
<td>EXAMPLE</td>
<td>+00000080</td>
<td>EXAMPLE</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>12</td>
<td>IBMPMINV</td>
<td>+0000040E</td>
<td>IBMPDUMP</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>13</td>
<td>CEEBV011</td>
<td>+00000008</td>
<td>CEEPLPKA</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
<tr>
<td>14</td>
<td>CEEBBEXT</td>
<td>+00000186</td>
<td>CEEPLPKA</td>
<td>11A3D336</td>
<td>114A4E38</td>
<td>11A3D336</td>
<td>Call</td>
</tr>
</tbody>
</table>

DSA DSA Addr E Addr PU Addr PU Offset Comp Date Compile Attributes
1 11A3DE50 114A4E38 11A3DE50 +000002AE 20061214 LIBRARY EBCDIC HFP
2 11A3DD70 112000D0 112000D0 +000000D2 20070131 ENT PL/I EBCDIC HFP
3 11A3DBD8 114A7B98 114A7B98 +000002A2 20061214 LIBRARY EBCDIC HFP
4 11A3D978 114A62E8 114A62E8 +000004DC 20061214 LIBRARY EBCDIC HFP
5 11A3ABA5 114A062E 114A062E +00000132 20061214 LIBRARY EBCDIC HFP
6 11A3A858 112C3120 112C3120 +000017D0 20061215 CEL
7 11A3A6B8 114A062E 114A062E +00000090 20061214 LIBRARY EBCDIC HFP
8 11A3A570 114A062E 114A062E +00000082 20061214 LIBRARY EBCDIC HFP
9 11A3A4A8 114A062E 114A062E +00000100 20070131 ENT PL/I EBCDIC HFP
10 11A3A3B8 114A062E 114A062E +00000320 20070131 ENT PL/I EBCDIC HFP
11 11A3A1B8 114A062E 114A062E +000001B6 20061214 LIBRARY EBCDIC HFP
12 11A3A0F0 114A062E 114A062E +00000202 20061214 LIBRARY EBCDIC HFP
13 11A3A030 112C3120 112C3120 +0000186 20061215 CEL

Condition Information for Active Routines

Condition Information for (DSA address 11A3A6B8)

CIB Address: 11A3B178

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: Entry: IBMBERRI Statement: Offset: +000000AA

Storage dump near condition, beginning at location: 114AAAB2

Figure 129. 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 9. The traceback information in the dump shows that the exception occurred following statement 16.

---

Figure 129. Sections of the Language Environment dump (Part 2 of 2)
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 Run-Time Messages.

3. Locate statement 14 in the routine in Figure 128 on page 337. 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 128 on page 337. Use this offset to find the I value at the time of the dump. In this example, the offset is X'8E'.

2. Now find offset X'8E' from the start of the stack frame for the entry EXAMPLE in Figure 129 on page 338. The block located at this offset contains the value that exceeded the array range, X'B' or 11.

Calling a nonexistent subroutine

Figure 130 on page 341 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) run-time option to generate a traceback.
Figure 130. Example of calling a nonexistent subroutine (Enterprise PL/I)

Figure 131 on page 342 shows the traceback and condition information sections from the dump.
Information for enclave EXAMPLE1

Traceback:

<table>
<thead>
<tr>
<th>DSA Addr</th>
<th>E Addr</th>
<th>PU Addr</th>
<th>PU Offset</th>
<th>Comp Date</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0C13DA70</td>
<td>0BBA4E30</td>
<td>0BBA4E30</td>
<td>+000002AE</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>0C13D990</td>
<td>0B9008A8</td>
<td>0B9008A8</td>
<td>+00000D2</td>
<td>20070131</td>
<td>ENT PL/I EBCDIC HFP</td>
</tr>
<tr>
<td>0C13D7F8</td>
<td>0BBA7B98</td>
<td>0BBA7B98</td>
<td>+000002A2</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>0C13D628</td>
<td>0BBAF390</td>
<td>0BBAF390</td>
<td>+000004DC</td>
<td>20061214</td>
<td>LIBRARY EBCDIC HFP</td>
</tr>
<tr>
<td>0C13D598</td>
<td>0BB062E8</td>
<td>0BB062E8</td>
<td>+00000132</td>
<td>20061214</td>
<td>LIBRARY</td>
</tr>
<tr>
<td>0C13D598</td>
<td>0BB062E8</td>
<td>0BB062E8</td>
<td>+00000132</td>
<td>20061214</td>
<td>LIBRARY</td>
</tr>
<tr>
<td>0C13D598</td>
<td>0BB062E8</td>
<td>0BB062E8</td>
<td>+00000132</td>
<td>20061214</td>
<td>LIBRARY</td>
</tr>
<tr>
<td>0C13D598</td>
<td>0BB062E8</td>
<td>0BB062E8</td>
<td>+00000132</td>
<td>20061214</td>
<td>LIBRARY</td>
</tr>
<tr>
<td>0C13D598</td>
<td>0BB062E8</td>
<td>0BB062E8</td>
<td>+00000132</td>
<td>20061214</td>
<td>LIBRARY</td>
</tr>
<tr>
<td>0C13D598</td>
<td>0BB062E8</td>
<td>0BB062E8</td>
<td>+00000132</td>
<td>20061214</td>
<td>LIBRARY</td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Condition Information for _ON_Begin_7_Blk_2 (DSA address 0C13A3B8)

CIB Address: 0C13AD98

Current Condition:

CEE3201S The system detected an operation exception (System Completion Code=0C1).

Location:

Program Unit: _ON_Begin_7_Blk_2
Statement: Offset: -0B9009A8
Possible Bad Branch: Statement: 12 Offset: +000001AE

Machine State:

ILC..... 0002 Interruption Code..... 0001
PSW..... 078D0600 80000002
GPR0..... 00000000_0C13A3B8 GPR1..... 00000000_0B9008A8 GPR2..... 00000000_0B911768 GPR3..... 00000000_0B9009E2
GPR4..... 00000000_0C13A0D8 GPR5..... 00000000_00000000 GPR6..... 00000000_0B900DA0 GPR7..... 00000000_00000000
GPR8..... 00000000_0B911648 GPR9..... 00000000_00000008 GPR10.... 00000000_0C13A0B0 GPR11.... 00000000_0B900F1C
GPR12.... 00000000_0B9129B0 GPR13.... 00000000_0C13A3B8 GPR14.... 00000000_8B900A58 GPR15.... 00000000_00000000
FPC...... F0000000
FPR0..... 26100000 00000000 FPR1..... 00000000 00000000
FPR2..... 18000000 00000000 FPR3..... 00000000 00000000
FPR4..... 00000000 00000000 FPR5..... 00000000 00000000
FPR6..... 00000000 00000000 FPR7..... 00000000 00000000
FPR8..... 00000000 00000000 FPR9..... 00000000 00000000
FPR10.... 00000000 00000000 FPR11.... 00000000 00000000
FPR12.... 00000000 00000000 FPR13.... 00000000 00000000
FPR14.... 00000000 00000000 FPR15.... 00000000 00000000

Storage dump near condition, beginning at location: 00000000
+0000000 00000000 Inaccessible storage.

Figure 131. 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 Run-Time 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'0B9009A8' 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 130 on page 341]. 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 132 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.

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 133 shows this output.

The routine in Figure 132 was compiled with the LIST compiler option, which generated the object code listing shown in Figure 134 on page 345.
Figure 134. Object code listing from example Enterprise PL/I routine

Figure 135 on page 346 shows the Language Environment dump for routine SAMPLE.
PLIDUMP called from error ON-unit 03/13/10 4:02:34 PM

Information for enclave SAMPLE

Information for thread 8000000000000000

Traceback:

DSA Entry E Offset Statement Load Mod Program Unit Service Status
1 IBMPDUMP +000002AE IBMPEV11 PQ78306 Call
2 _ON_Begin_4_Blk_2 +000000D4 6 SAMPLE SUB1 Call
3 IBMPDUMP +000002A2 IBMPEV11 PQ76426 Call
4 IBMPEONR +0000040C IBMPEV11 LE19BAS Call
5 CEEEV011 +00000312 IBMPEV11 CEEEV011 Call
6 CEEHDSP +000017D0 CEEPLPKA CEEHDSP D1908 Call
7 SUB1 +00000170 CEEPLPKA CEEHDSP D1908 Call
8 SAMPLE +000001C6 27 SAMPLE SUB1 Call
9 IBMPMINV +0000045E IBMPEV11 IBMMPINV Call
10 CEEEV011 +000002D2 IBMPEV11 CEEEV011 Call

DSA DSA Addr E Addr PU Addr PU Offset Comp Date Compile Attributes
1 11A3ED08 11A54E38 11A54E38 +000002AE 20061214 LIBRARY EBCDIC HFP
2 11A3EB00 11200340 112000D0 +00000344 20070131 ENT PL/I EBCDIC HFP
3 11A3E838 11488B98 11488B98 +000002A2 20061214 LIBRARY EBCDIC HFP
4 11A3E848 11480390 11480390 +000004DC 20061214 LIBRARY EBCDIC HFP
5 11A3E7B8 114072E8 114072E8 +00000132 20061214 LIBRARY
6 11A3E898 112C4238 112C4238 +000004DE 20061214 LIBRARY EBCDIC HFP
7 11A3EBB8 112000B8 112000B8 +000001C6 20070131 ENT PL/I EBCDIC HFP
8 11A3E868 112004B8 112000B8 +000005B2 20070131 ENT PL/I EBCDIC HFP
9 11A3E800 114DE990 114DE990 +000004DE 20061214 LIBRARY
10 11A3B0F0 114072E8 114072E8 +00000202 20061214 LIBRARY
11 11A3B030 11292208 11292208 +000001B6 20061215 CEL

Condition Information for Active Routines

Condition Information for SUB1 (DSA address 11A3B538)
CIB Address: 11A3BFB8
Current Condition:
IBM0281S A prior condition was promoted to the ERROR condition.
Original Condition:
CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).
Location:
Program Unit: SUB1 Entry: SUB1 Statement: 27 Offset: +000001C6
Machine State:
ILC... 0002 Interruption Code..... 0009
PSW..... 07B02600 912009B8
GPR0..... 000000_00000000 GPR1..... 000000_00000000 GPR2..... 000000_00000000 GPR3..... 000000_00000000 GPR4..... 000000_00000000 GPR5..... 000000_00000000 GPR6..... 000000_00000000 GPR7..... 000000_00000000 GPR8..... 000000_00000000 GPR9..... 000000_00000000 GPR10.... 000000_00000000 GPR11.... 000000_00000000 GPR12.... 000000_00000000 GPR13.... 000000_00000000 GPR14.... 000000_00000000 GPR15.... 000000_00000000
FPC...... F0000000
FPR0..... 26100000 00000000 FPR1..... 00000000 00000000 FPR2..... 00000000 00000000 FPR3..... 00000000 00000000 FPR4..... 00000000 00000000 FPR5..... 00000000 00000000 FPR6..... 00000000 00000000 FPR7..... 00000000 00000000 FPR8..... 00000000 00000000 FPR9..... 00000000 00000000 FPR10.... 00000000 00000000 FPR11.... 00000000 00000000 FPR12.... 00000000 00000000 FPR13.... 00000000 00000000 FPR14.... 00000000 00000000 FPR15.... 00000000 00000000

Storage dump near condition, beginning at location: 112D0286
+000000_112D0286 5B20014C 5B202000 5B202000 8E800200 10821849 5B202058 50402000 4400C1AC |...J...............b....J.& ....A.| :

Figure 135. Language Environment dump from example Enterprise PL/I routine (Part 1 of 2)
Figure 135. Language Environment dump from example Enterprise PL/I routine (Part 2 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 Run-Time 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 134 on page 345. 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.
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 run-time 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 run-time 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 run-time 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 DFHDCT 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 run-time messages

Under CICS, Language Environment run-time 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 19910916145313 CEE3250C The System or User ABEND AEI0 was issued. P039UTV9 19910916145313 From program unit UT9COVERI at entry point UT9COVERIT +0000011E at P039UTV9 19910916145313 at offset address 0000011E.
```
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 136 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 42.

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 run-time 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 Run-Time 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 JOB35555 IEF450 XCEPI03 GD CEPI03 - 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 Run-Time 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 Run-Time 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 137) just formats the heading line for each trace point and is laid out in a similar way to CICS trace entries.

Figure 137. 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 138 on page 354.
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 run-time 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 23 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 47 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>DFHAP1200I 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 48 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>
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 run-time option TERMTHDACT(MSG).

Table 49. 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. Resource backout was successful.</td>
<td>User’s terminal</td>
<td>CICS</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 UT9CVERI term P021 backout successful.</td>
<td>Transient data queue CSMT</td>
<td>CICS</td>
</tr>
<tr>
<td>P021UTV9 091156 143516 CEE3250C The System or User Abend AEI0 was issued.</td>
<td>Transient data queue CESE</td>
<td>Language Environment</td>
</tr>
</tbody>
</table>

Displaying and modifying run-time options with the CLER transaction

The CICS transaction (CLER) allows you to display all the current Language Environment run-time options for a region, and to also have the capability to modify a subset of these options. The CLER transaction can be used to:

- Display the current run-time options in effect for the region.
- Modify the following subset of the region run-time options:
  - ALL31(ON|OFF)
  - CBLPUSHPOP(ON|OFF)
  - CHECK(ON|OFF)
  - INFORMSGFILTER(ON|OFF)
  - RTPOPTS(ON|OFF)
  - RPTSTG(ON|OFF)
  - TERMTHDACT(QUIET|MSG|TRACE|DUMP|UAONLY|UATRACE|UADUMP|UAIMM)
  - TRAP(ON|OFF)
- Write the current region run-time options to the CESE queue for printing.

The CLER transaction is conversational; it presents the user with commands for the terminal display. The run-time 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.

Note: If the run-time 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 run-time option ALL31 is set to OFF or CBLPSHPOP, RPTOPTS, and RPTSTG are set to ON.

To send the run-time option report to the CESE queue for output display or printing, press PF10 on the panel which displays the run-time 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++
- Language Environment run-time 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 z/OS XL C/C++ User’s Guide and z/OS XL C/C++ Programming Guide.

Using Language Environment run-time options

Several run-time options affect debugging in Language Environment. The TEST run-time 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 run-time options affect debugging. For a more detailed discussion of these run-time options, see z/OS Language Environment Programming Reference.

CEEDUMP

Specifies options to control the processing of the Language Environment dump report.
**DYNDUMP**  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.

**HEAPCHK**  Determines whether additional heap check tests are performed.

**INFOMSGFILTER**  Filters user specified informational messages from stderr.  
*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.

**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**  Causes a report to be produced which contains the run-time options in effect. See “Determining run-time options in effect” below.

**RPTSTG**  Generates a report of the storage used by an enclave. See “Controlling storage allocation” on page 363.

**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 run-time 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 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 run-time results can be unpredictable.

---

**Determining run-time options in effect**

The run-time 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 run-time options, and indicates where they were set. Figure 139 on page 363 shows a sample options report.
Controlling storage allocation

The following run-time options control storage allocation:

- HEAP64
- HEAPPOLS
- HEAPPOLLS64
- 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) run-time 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 run-time options for future runs. Figure 140 on page 364 shows a sample storage report.
Storage Report for Enclave main Wed Jan 21 16:52:36 2009
Language Environment V01 R12.00

<table>
<thead>
<tr>
<th>Statistics Type</th>
<th>Initial Size</th>
<th>Increment Size</th>
<th>Max Used by All Concurrent Threads</th>
<th>Largest Used by Any Thread</th>
<th>Num of Increments Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACK64</td>
<td>1M</td>
<td>1M</td>
<td>1M</td>
<td>1M</td>
<td>0</td>
</tr>
<tr>
<td>THREADSTACK64</td>
<td>1M</td>
<td>1M</td>
<td>0M</td>
<td>0M</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistics Type</th>
<th>Initial Size</th>
<th>Increment Size</th>
<th>Max Used by All Concurrent Threads</th>
<th>Largest Used by Any Thread</th>
<th>Num of Increments Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>64bit User HEAP</td>
<td>1M</td>
<td>1M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31bit User HEAP</td>
<td>32768</td>
<td>32768</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24bit User HEAP</td>
<td>4096</td>
<td>4096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64bit Library HEAP</td>
<td>1M</td>
<td>1M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31bit Library HEAP</td>
<td>16384</td>
<td>8192</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 140. 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: 0
Requests greater than the largest cell size: 0

HEAPPools Summary:
<table>
<thead>
<tr>
<th>Specified Element Extent</th>
<th>Cells Per Extents</th>
<th>Maximum Cells Allocated</th>
<th>Cells Used</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Size</td>
<td>Size</td>
<td>Percent</td>
<td>Extent</td>
<td>409</td>
</tr>
<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 140. 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 2</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>240</td>
<td>9-16 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>17-24 227</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25-32 1</td>
</tr>
<tr>
<td>3</td>
<td>128</td>
<td>125</td>
<td>33-40 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>41-48 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>49-56 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>57-64 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>65-72 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>73-80 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>81-88 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>89-96 88</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>105-112 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>113-120 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>121-128 4</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>53</td>
<td>129-136 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>137-144 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>145-152 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>153-160 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>161-168 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>169-176 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>177-184 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>185-192 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>193-200 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>201-208 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>209-216 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>217-224 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>225-232 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>233-240 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>249-256 3</td>
</tr>
<tr>
<td>5.1</td>
<td>1024</td>
<td>2</td>
<td>1024 2</td>
</tr>
<tr>
<td>5.2</td>
<td>1024</td>
<td>2</td>
<td>1024 2</td>
</tr>
<tr>
<td>5.3</td>
<td>1024</td>
<td>0</td>
<td>1024 0</td>
</tr>
<tr>
<td>6</td>
<td>2048</td>
<td>2</td>
<td>2048 2</td>
</tr>
<tr>
<td>7</td>
<td>3072</td>
<td>2</td>
<td>3072 2</td>
</tr>
<tr>
<td>8</td>
<td>4096</td>
<td>1</td>
<td>4096 1</td>
</tr>
<tr>
<td>9</td>
<td>8192</td>
<td>0</td>
<td>8192 0</td>
</tr>
<tr>
<td>10</td>
<td>16384</td>
<td>0</td>
<td>16384 0</td>
</tr>
<tr>
<td>11</td>
<td>32768</td>
<td>0</td>
<td>32768 0</td>
</tr>
<tr>
<td>12</td>
<td>65536</td>
<td>0</td>
<td>65536 0</td>
</tr>
</tbody>
</table>

Requests greater than the largest cell size: 0

Figure 140. 64-bit storage report (Part 3 of 4)
Storage statistics for AMODE 64 applications

The statistics for initial and incremental allocations of storage types that have a corresponding run-time option differ from the run-time 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 run-time options in [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 run-time options.
- Increment size—the size of each incremental stack area made available, as determined by the increment portion of the corresponding run-time 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.

### HEAPPOOLS64 Summary:

<table>
<thead>
<tr>
<th>Specified Cell Size</th>
<th>Element Size</th>
<th>Cells Per Extent</th>
<th>Extents Allocated</th>
<th>Maximum Cells Used</th>
<th>Cells In Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>32</td>
<td>4000</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>48</td>
<td>2000</td>
<td>1</td>
<td>226</td>
<td>225</td>
</tr>
<tr>
<td>128</td>
<td>144</td>
<td>700</td>
<td>1</td>
<td>83</td>
<td>77</td>
</tr>
<tr>
<td>256</td>
<td>272</td>
<td>350</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1040</td>
<td>34</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1024</td>
<td>1040</td>
<td>34</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>2064</td>
<td>50</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3072</td>
<td>3088</td>
<td>50</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4096</td>
<td>4112</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>8208</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>16400</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>32784</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>65536</td>
<td>65552</td>
<td>5</td>
<td>0</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 140. 64–bit storage report (Part 4 of 4)*
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.

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 run-time 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 run-time option.
- Increment size—the minimum incremental allocation, as specified by the corresponding run-time 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 run-time 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 502.

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>API</th>
<th>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 run-time options

The following Language Environment run-time options can affect your routine’s exception handling behavior:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</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.  
  During normal operation, you should use TRAP(ON) when running your applications. |
TRAP(OFF)

Disables the Language Environment condition handler from handling abends and program checks/interrupts. ESPIE is not issued with TRAP(OFF).

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 run-time 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 run-time 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 372).

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>Function Name</th>
<th>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.

![Figure 141. 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 run-time 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 run-time messages and corrective information, see z/OS Language Environment Run-Time 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 run-time 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 run-time options and callable services. (See "Controlling storage allocation" on page 363 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 run-time messages point to the nature of the error. The run-time messages offer the most efficient corrective action. To help you analyze errors and determine the most useful method to fix the problem, Table 50 on page 373 lists common error symptoms, possible causes, and programmer responses.
### Table 50. 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 run-time message appears</td>
<td>Condition raised in routine</td>
<td>For any messages you receive, read the Programmer Response. For information about</td>
</tr>
<tr>
<td></td>
<td></td>
<td>message structure, see “Interpreting run-time messages” below.</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 Run-Time</td>
</tr>
<tr>
<td></td>
<td>• The assembler user exit requested an abend for an unhandled condition of</td>
<td>Messages.</td>
</tr>
<tr>
<td></td>
<td>severity ≥2</td>
<td>Check for a subsystem-generated abend or a user-specified abend.</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</td>
</tr>
<tr>
<td></td>
<td>• An unhandled software-raised condition occurred</td>
<td>codes section of z/OS Language Environment Run-Time Messages.</td>
</tr>
<tr>
<td></td>
<td>• The assembler user exit requested an abend for an unhandled condition of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>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 book of the operating system.</td>
</tr>
<tr>
<td>No response (wait/loop)</td>
<td>Application logic failure</td>
<td>Check routine logic.</td>
</tr>
<tr>
<td>Unexpected message (message received was not</td>
<td>Condition caused by something related to current service</td>
<td>Generate a traceback using cdump() or ctrace().</td>
</tr>
<tr>
<td>recent service)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incorrect output</td>
<td>Incorrect file definitions, storage overlay, incorrect routine mask setting,</td>
<td>Correct the appropriate parameters.</td>
</tr>
<tr>
<td></td>
<td>references to uninitialized variables, data input errors, or application</td>
<td></td>
</tr>
<tr>
<td></td>
<td>routine logic error</td>
<td></td>
</tr>
<tr>
<td>No output</td>
<td>Incorrect ddname or file definitions</td>
<td>Correct the appropriate parameters.</td>
</tr>
<tr>
<td>Nonzero return code from enclave</td>
<td>The return code was issued by the application routine</td>
<td>Check the application for the meaning of the return code.</td>
</tr>
</tbody>
</table>

### Interpreting run-time messages

The first step in debugging your routine is to look up any run-time messages. Run-time messages are written to the C stderr stream. Run-time 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 run-time 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)”.

374  z/OS V1R13.0 Language Environment Debugging Guide
Language Environment messages can appear even though you made no explicit calls to Language Environment services. C/C++ run-time 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 run-time 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 Run-Time 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 run-time 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 hhh 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++ Run-Time Library Reference.

User abends

If you receive a Language Environment abend code, see z/OS Language Environment Run-Time 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 run-time option is used in combination with the TERMTHDACT run-time 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 394 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 run-time 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, UAONLY, 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>
<tr>
<td>Suboption</td>
<td>Level of Information</td>
<td>Destination</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------</td>
<td>-------------</td>
</tr>
<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(CURRENT)
Considerations for setting TERMTHDACT options

Review the following considerations before setting TERMTHDACT run-time 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=CELAICT+????, ABEND=U4039,REASON=00000000

  - For UAIMM, 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=CELAICT+????, ABEND=500C9,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 476.

Understanding the Language Environment dump

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

Figure 144 on page 382 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 "Sections of the Language Environment dump" on page 390.
The CEE3DMP was generated by the C program CELQSAMP shown in Figure 142.

CELQSAMP uses the DLL CELQDLL shown in Figure 143 on page 381.

Figure 142. 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 142. The C program CELQSAMP (AMODE 64) (Part 2 of 2)

('').F.('F',.Q,'D','L',' ','W','S','A');

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 143. 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 390.


[3]  Information for enclave main

[4]  Information for thread 253E019000000000

[5]  Traceback:

<table>
<thead>
<tr>
<th>DSA Entry</th>
<th>E Addr</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>CELQDLL</td>
<td>CELQDLL</td>
<td>D1908</td>
<td>Call</td>
</tr>
<tr>
<td>1</td>
<td>+00000000</td>
<td>CELQIB</td>
<td>CELHDSP</td>
<td>CEEHDSP</td>
<td>01908</td>
<td>Call</td>
</tr>
<tr>
<td>2</td>
<td>+0000094E</td>
<td>CELQIB</td>
<td>CELQDLL</td>
<td>CEEOSIGJ</td>
<td>01908</td>
<td>Call</td>
</tr>
<tr>
<td>3</td>
<td>+0000024E</td>
<td>CELQIB</td>
<td>CELHDSP</td>
<td>CELQMOD</td>
<td>01908</td>
<td>Call</td>
</tr>
<tr>
<td>4</td>
<td>+18003E4B</td>
<td>CELQIB</td>
<td>CELQMOD</td>
<td>CELQMOD</td>
<td>01908</td>
<td>Call</td>
</tr>
<tr>
<td>5</td>
<td>+0000024E</td>
<td>CELQIB</td>
<td>CELHDSP</td>
<td>CELQMOD</td>
<td>01908</td>
<td>Call</td>
</tr>
<tr>
<td>6</td>
<td>+0000004E</td>
<td>CELQDLL</td>
<td>CELQDLL</td>
<td>CELQDLL</td>
<td>1.4.f</td>
<td>Exception</td>
</tr>
<tr>
<td>7</td>
<td>+00000468</td>
<td>9B</td>
<td>CELQSAMP</td>
<td>CELQSAMP</td>
<td>1.2.d</td>
<td>Call</td>
</tr>
<tr>
<td>8</td>
<td>+000134C</td>
<td>CELQIB</td>
<td>CELQINIT</td>
<td>CELQINIT</td>
<td>01908</td>
<td>Call</td>
</tr>
</tbody>
</table>

DSA DSA Addr E Addr PU Addr PU Offset Comp Date Compile Attributes
1 00000001082FAAC0 00000000251B6060 00000000 20061215 CEL POSIX XPLINK EBCDIC HFP
2 00000001082FD3E0 000000002504AAB0 00000000 20070109 CEL POSIX XPLINK EBCDIC HFP
3 00000001082FDDE0 00000000251C9480 00000000 20061215 CEL POSIX XPLINK EBCDIC HFP
4 00000001082FE020 00000000253D11F8 000000001B032E48 20061215 CEL POSIX XPLINK EBCDIC HFP
5 00000001082FEE40 00000000251C9480 00000000 20061215 CEL POSIX XPLINK EBCDIC HFP
6 00000001082FF080 000000002575B5A0 0000000000000000 ******** 20070117 C/C++ POSIX XPLINK EBCDIC IEEE
7 00000001082FF180 00000000250000D8 0000000000000000 ******** 20070117 C/C++ POSIX XPLINK EBCDIC IEEE
8 00000001082FF280 0000000025005010 0000000025005010 00000134C 20061215 CEL POSIX XPLINK EBCDIC HFP

Fully Qualified Names

<table>
<thead>
<tr>
<th>DSA Entry</th>
<th>Program Unit</th>
<th>Load Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 div_zero</td>
<td>PLPSC://POSIX.CRTL.C(CELQDLL)'</td>
<td>CELQDLL</td>
</tr>
<tr>
<td>7 main</td>
<td>PLPSC://POSIX.CRTL.C(CELQSAMP)'</td>
<td>CELQSAMP</td>
</tr>
</tbody>
</table>

[6]  Condition Information for Active Routines

Condition Information for PLPSC://POSIX.CRTL.C(CELQDLL)' (DSA address 00000001082FF080)
CIB Address: 000000100027E800

Current Condition: CEE0198S The termination of a thread was signaled due to an unhandled condition.

Original Condition: CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).

Location:
Program Unit: PLPSC://POSIX.CRTL.C(CELQDLL)' Entry: div_zero Statement: 15 Offset: +000004E

Machine State:
ILC..... 0002 Interruption Code..... 0009
PSW..... 0785240180000000 0000000000000000
GPR0..... 0000000000000000 GPR1..... 0000000000000000 GPR2..... 0000000000000000
GPR3..... 0000000000000000 GPR4..... 0000000000000000 GPR5..... 0000000000000000
GPR6..... 0000000000000000 GPR7..... 0000000000000000
GPR8..... 0000000000000000 GPR9..... 0000000000000000 GPR10.... 0000000000000000
GPR11.... 0000000000000000 GPR12.... 0000000000000000 GPR13.... 0000000000000000
GPR14.... 0000000000000000 GPR15.... 0000000000000000

Figure 144. Example dump using CEE3DMP (AMODE 64) (Part 1 of 9)
Figure 144. Example dump using CEE3DMP (AMODE 64) (Part 2 of 9)
Saved Registers:
GPR0..... 0000001087FD3E0 GPR1..... 0000001087FD3E0 GPR2..... 0000001087FD3E0 GPR3..... 0000001087FD3E0
GPR4..... 0000000000000000 GPR5..... 0000000000000000 GPR6..... 0000000000000000 GPR7..... 0000000000000000
GPR8..... 0000000000000000 GPR9..... 0000000000000000 GPR10..... 0000000000000000 GPR11..... 0000000000000000
GPR12..... 0000000000000000 GPR13..... 0000000000000000 GPR14..... 0000000000000000 GPR15..... 0000000000000000

GPRREG STORAGE:
Storage around GPR0 is invalid.
Storage around GPR1 is invalid.

[8] Control Blocks for Active Routines:

DSA for CEEOSIG: 0000000000000000
+000000 00000001082FD3E0 R5..... 0000000000000000 R6..... 0000000000000000
+000018 082FD3E0 R8..... 0000000000000000 R9..... 0000000000000000
+000030 082FD3FC0 R10..... 0000000000000000 R12..... 0000000000000000
+000048 082FD3E80 R14..... 0000000000000000 R15..... 0000000000000000
+000060 reserved. 0000000000000000 reserved. 0000000000000000

DSA for CELQHROD: 0000000000000000
+000000 00000001082FD3E0 R5..... 0000000000000000 R6..... 0000000000000000
+000018 082FD3E0 R8..... 0000000000000000 R9..... 0000000000000000
+000030 082FD3FC0 R10..... 0000000000000000 R12..... 0000000000000000
+000048 082FD3E80 R14..... 0000000000000000 R15..... 0000000000000000
+000060 reserved. 0000000000000000 reserved. 0000000000000000

DSA for CELQHROD: 0000000000000000
+000000 00000001082FD3E0 R5..... 0000000000000000 R6..... 0000000000000000
+000018 082FD3E0 R8..... 0000000000000000 R9..... 0000000000000000
+000030 082FD3FC0 R10..... 0000000000000000 R12..... 0000000000000000
+000048 082FD3E80 R14..... 0000000000000000 R15..... 0000000000000000
+000060 reserved. 0000000000000000 reserved. 0000000000000000

Control Blocks for Active Routines:

DSA for CELQHROD: 0000000000000000
+000000 00000001082FD3E0 R5..... 0000000000000000 R6..... 0000000000000000
+000018 082FD3E0 R8..... 0000000000000000 R9..... 0000000000000000
+000030 082FD3FC0 R10..... 0000000000000000 R12..... 0000000000000000
+000048 082FD3E80 R14..... 0000000000000000 R15..... 0000000000000000
+000060 reserved. 0000000000000000 reserved. 0000000000000000

DSA for CELQHROD: 0000000000000000
+000000 00000001082FD3E0 R5..... 0000000000000000 R6..... 0000000000000000
+000018 082FD3E0 R8..... 0000000000000000 R9..... 0000000000000000
+000030 082FD3FC0 R10..... 0000000000000000 R12..... 0000000000000000
+000048 082FD3E80 R14..... 0000000000000000 R15..... 0000000000000000
+000060 reserved. 0000000000000000 reserved. 0000000000000000

DSA for CELQHROD: 0000000000000000
+000000 00000001082FD3E0 R5..... 0000000000000000 R6..... 0000000000000000
+000018 082FD3E0 R8..... 0000000000000000 R9..... 0000000000000000
+000030 082FD3FC0 R10..... 0000000000000000 R12..... 0000000000000000
+000048 082FD3E80 R14..... 0000000000000000 R15..... 0000000000000000
+000060 reserved. 0000000000000000 reserved. 0000000000000000

CIB for div_zero(00000000000000000000000000000000)
+000000 00000000000000000000000000000000 CIB 00000000000000000000000000000000
+010000 00000000000000000000000000000000 CIB 00000000000000000000000000000000
+020000 00000000000000000000000000000000 CIB 00000000000000000000000000000000
+030000 00000000000000000000000000000000 CIB 00000000000000000000000000000000
+040000 00000000000000000000000000000000 CIB 00000000000000000000000000000000
+050000 00000000000000000000000000000000 CIB 00000000000000000000000000000000

Figure 144. Example dump using CEE3DMP (AMODE 64) (Part 3 of 9)
Figure 144. Example dump using CEE3DMP (AMODE 64) (Part 4 of 9)
Information for thread 253E10A000000001

Traceback:

<table>
<thead>
<tr>
<th>DSA Entry</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CEEOPML2</td>
<td>+00000000</td>
<td>CELQLIB</td>
<td>CEEOPML2</td>
<td>D1908 Call</td>
</tr>
<tr>
<td>2</td>
<td>thread_func</td>
<td>+0000005A</td>
<td>24</td>
<td>CELQSAMP</td>
<td>1.2.d Call</td>
</tr>
<tr>
<td>3</td>
<td>CELQPCMM</td>
<td>+00000DEA</td>
<td>CELQLIB</td>
<td>CELQPCMM</td>
<td>D1908 Call</td>
</tr>
</tbody>
</table>

DSA Addr E Addr PU Addr PU Offset Comp Date Compile Attributes

1 00000001111FF6F00 00000005256F6000 0000000025920B00 00000000 2005215 CEL POSIX XPLINK EBCDIC HFP
2 00000001111FF2C00 0000000000000000 0000000000000000 0000000000000000 C/C++ POSIX XPLINK EBCDIC IEEE
3 00000001111FF3C00 0000000108910290 0000000000000000 0000000000000000 20061214 CEL POSIX XPLINK EBCDIC HFP

Fully Qualified Names

DSA Entry Program Unit Load Module

2 thread_func PLPSC://POSIX.CRTL.C(CELQSAMP) CELQSAMP

GPR0..... 0000000000000001 GPR1..... 00000001113005B8 GPR2..... 0000000108358750 GPR3..... 00000001113005B8
GPR4..... 00000001111FF6F00 GPR5..... 00000000257669A0 GPR6..... 0000000108358750 GPR7..... 00000001113005B8
GPR8..... 00000001113005B8 GPR9..... 00000001113005B8 GPR10.... 0000000111401140 GPR11.... 00000001807D17D8
GPR12.... 00000001113005B8 GPR13.... 00000001113005B8 GPR14.... 00000001113005B8 GPR15.... 00000001807D17D8

GPREG STORAGE:

-0001 0000000000000000 Inaccessible storage.
+000F 0000000000000010 Inaccessible storage....

Control Blocks for Active Routines:

DSA for CEEOPML2: 00000001111FF760

+000000 R4....... 00000001111FFB60 R5....... 000000002576F000 R6....... 000000002576F000
+000018 R7....... 0000000025520604 R8....... 00000000250005B4 R9....... 000000002576F000
+000030 R10...... 0000000025520604 R11...... 00000000250005B4 R12...... 0000000025520604
+000048 R13...... 0000000025520604 R14...... 00000000250005B4 R15...... 0000000025520604

DSA for thread_func: 00000001111FFAC0

+000000 R4....... 00000001111FF3C0 R5....... 0000000108300070 R6....... 000000002576F000
+000018 R7....... 0000000025520604 R8....... 00000000250005B4 R9....... 000000002576F000
+000030 R10...... 0000000025520604 R11...... 00000000250005B4 R12...... 0000000025520604
+000048 R13...... 0000000025520604 R14...... 00000000250005B4 R15...... 0000000025520604

Storage for Active Routines:

DSA frame(00000001111FF6F00)

+0000 00000001111FF760 00000000111FF6F00 000000002576F000 0000000025520604 00000000250005B4 000000002576F000
+00018 00000001111FF3C0 00000000111401140 000000002576F000 0000000025520604 00000000250005B4 000000002576F000
+00030 00000001111FF760 00000000111FF6F00 000000002576F000 0000000025520604 00000000250005B4 000000002576F000

Figure 144. Example dump using CEE3DMP (AMODE 64) (Part 5 of 9)
Control Blocks Associated with the Thread:

```
+0000 00000001140138 00000000 00000000 00000000 00000000 |.................|
+0010 00000001140138 +0002AF 000000011401677 same as above
+0280 000000011401678 00000000 00000000 00000000 00000000 |.................|
```

Enclave Control Blocks:

```
00000001083A0430 00000000258C4000 253E019000000000 00000000 CELQDSNF
0000000108300050 00000001 main
```

WSA Addr Module Addr Thread ID Use Count Name

DLL Information:

```
0000000108FC0090 0000000108FC00B0 0000000108FC00D0 0000000108FC00F0
```

Mutex and Condition Variable Blocks (MCV+MHT+CHT)(00000001089100B8)

```
+0000 00000001089100B8 00000001089100C8 00000001089100D8 |.................|
+0280 00000001089100F8 0000000108910108 0000000108910118 |.................|
```

Header(00000001000068F8)

```
+0000 00000001000068F8 0000000100006908 0000000100006938 |.................|
+0010 0000000100006928 0000000100006958 0000000100006968 |.................|
+0020 0000000100006A40 0000000100006A60 0000000100006A80 |.................|
```

HEAPCHK Option Control Block (HCOP)(00000001089234D0)

```
+0000 00000001089234D0 00000001089234E0 00000001089234F0 0000000108923500 |.................|
```

HEAPCHK Element Table (HECEL) for Heapid 0000000100100138:

```
0000000108FC0090 0000000108FC00B0 0000000108FC00D0 0000000108FC00F0
```

Figure 144. Example dump using CEE3DMP (AMODE 64) (Part 6 of 9)
Figure 144. Example dump using CEE3DMP (AMODE 64) (Part 7 of 9)
Heap Storage Diagnostics
All storage has been freed.

Figure 144. Example dump using CEE3DMP (AMODE 64) (Part 8 of 9)
Sections of the Language Environment dump
The sections of the dump listed in Table 52 appear independently of the Language Environment-conforming languages used.

Table 52. Contents of the Language Environment dump - AMODE 64

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [1] Page Heading           | The page heading section appears on the top of each page of the dump and contains:  
   • CEE3DMP identifier  
   • Title For dumps generated as a result of an unhandled condition, the title is “Condition processing resulted in the Unhandled condition.”  
   • Product abbreviation of Language Environment  
   • Version number  
   • Release number  
   • Date  
   • Time  
   • Page number |
Table 52. 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>[2] CEE3845I CEEDUMP</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] 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 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 &quot;** NoName **&quot; 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 [XL C and XL C++] compiler options for AMODE 64 applications” on page 361.</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.</td>
</tr>
<tr>
<td></td>
<td>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>• Status: Routine status can be call or exception.</td>
</tr>
</tbody>
</table>
Table 52. 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><strong>[5] Traceback (continued)</strong></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 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>• Compile Date</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
</tbody>
</table>

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. |

**[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 |
|                           | • 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.
**Table 52. Contents of the Language Environment dump - AMODE 64 (continued)**

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [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. |
| [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 run-time options are set.  
  - If the POSIX run-time 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 run-time 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 run-time 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 run-time 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] Run-Time Options Report | Lists the Language Environment run-time options in effect when the routine was executed. |
### 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 run-time 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 run-time 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 377.

- **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 run-time environments. The following sections describe the recommended steps needed to generate a system dump in batch and z/OS UNIX shell run-time environments. Other methods may exist, but these are the recommended steps for generating a system dump. For details on setting Language Environment run-time options, see [z/OS Language Environment Programming Guide](#).
Steps for generating a system dump in a batch run-time environment

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

1. Specify run-time 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 377.

2. Decide whether to include a SYSMDUMP DD card or use the DYNDUMP run-time 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 run-time 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 run-time options, where suboption equals 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 377.
     - 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 run-time options:
     - export _CEE_RUNOPTS="termthdact(suboption),DYNDUMP(hlg,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 377.

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 run-time 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 CEEIPCSP 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
- Run-time Options
- Storage Management Control Blocks
- Condition Management Control Blocks
- Message Handler Control Blocks
- C/C++ Control Blocks

Format

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

Report Type Parameters:
- AUTH
- NTHREADS(value)
- SUM
- HEAP | STACK | SM
- HPT(number) [ HPTCB(address) ] [ HPTCELL(address) ] [ HPTLOC(location) ]
- CH
- MH
- CEEEDUMP
- 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)
- LAA(laa-address)

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 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.

**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
  - Run-time 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 heappools 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.

- **HPT(number) [ HPTCB (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 HEAPPOOL trace table, if available, printing only those entries for a given TCB address *(address)*.

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

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

- **31** Display entries located in virtual storage below the bar.
- **64** Display entries located in virtual storage above the bar.
- **ALL** Display entries located in virtual storage below or above the bar.

**Notes:**
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. 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++ run-time 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.

DETa1
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 415.

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++ 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 LAA.

DSA(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(tcb-address)
specifies the address of the TCB. If not specified, the TCB address may be obtained from the CAA or the CVT.

LAA(laa-address)
specifies the address of the LAA. If not specified, the LAA address may be obtained from the TCB or the PSA.

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.

Examples
For examples of the output produced by LEDATA and explanation of the content, refer to "Understanding the Language Environment IPCS VERBEXIT LEDATA output" on page 401.
Understanding the Language Environment IPCS VERBEXIT LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of the Language Environment run-time environment control blocks from a system dump. Figure 145 on page 402 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) run-time option when running the program CELQSAMP in Figure 142 on page 380.

“Sections of the Language Environment LEDATA VERBEXIT formatted output” on page 411 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 411.
Figure 145. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 10)
Figure 145. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 2 of 10)
Figure 145. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 3 of 10)
Figure 145. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 4 of 10)
Figure 145. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 5 of 10)
Heap Storage Control Blocks

- ENSQ: 00000001_00100108
- EYE_CATCHER: ENSQ HEAPALLOC_VAL: 00000000
- HEAPFREE_VAL: 00000000 DSAALLOC_VAL: 00000000
- IPT_TOKEN: 0000000000000002 0000000000000001 0000000000000001 0000000000000000
- HTD: 00000000 00000001 00000000 00000000 00000000 00000000 00000000 00000000
- HEAPID: 00000000 00000001 PREV: 00000000 00000001
- HEAPFREE: 00000000 00000001 GETMAINS: 00000000 00000001
- CURR_ALLOC: 00000000 00000001 GET_REQ: 00000000 00000001
- BYTES_ALLOC: 00000000 00000001
- INITSIZE: 00000000 00000001 INCRSIZE: 00000000 00000001
- HPCQ: 00000001_00100138
- FIRST: 00000000 00000001
- LAST: 00000000 00000001
- GETMAINS: 00000000 00000001
- SEG_LEN: 00000000 00000001
- HPCQ: 00000001_00100138
- FIRST: 00000000 00000001
- LAST: 00000000 00000001
- GETMAINS: 00000000 00000001
- SEG_LEN: 00000000 00000001
- HPCQ: 00000001_00100138
- FIRST: 00000000 00000001
- LAST: 00000000 00000001
- GETMAINS: 00000000 00000001
- SEG_LEN: 00000000 00000001
- HPCQ: 00000001_00100138
- FIRST: 00000000 00000001
- LAST: 00000000 00000001
- GETMAINS: 00000000 00000001
- SEG_LEN: 00000000 00000001

User Heap64 Control Blocks

- HPCQ: 00000001_00100138
- FIRST: 00000000 00000001
- LAST: 00000000 00000001
- GETMAINS: 00000000 00000001
- SEG_LEN: 00000000 00000001

Free Storage Tree for Heap Segment 00000001

To display entire segment: IP LIST 000000000010030000 LEN(*'000000000010080000') ASID(*'00028' )

- 000000000010083000 LEN(*'000000000010080000') ASID(*'00028' )
- 000000000010083000 LEN(*'000000000010080000') ASID(*'00028' )
- 000000000010083000 LEN(*'000000000010080000') ASID(*'00028' )
- 000000000010083000 LEN(*'000000000010080000') ASID(*'00028' )
- 000000000010083000 LEN(*'000000000010080000') ASID(*'00028' )
- 000000000010083000 LEN(*'000000000010080000') ASID(*'00028' )

Map of Heap Segment 00000001

To display entire segment: IP LIST 00000000001008300000 LEN(*'000000000010080000') ASID(*'00028' )

- 00000000001008300000 LEN(*'000000000010080000') ASID(*'00028' )
- 00000000001008300000 LEN(*'000000000010080000') ASID(*'00028' )
- 00000000001008300000 LEN(*'000000000010080000') ASID(*'00028' )
- 00000000001008300000 LEN(*'000000000010080000') ASID(*'00028' )
- 00000000001008300000 LEN(*'000000000010080000') ASID(*'00028' )
- 00000000001008300000 LEN(*'000000000010080000') ASID(*'00028' )

Chapter 12. Using Language Environment AMODE 64 debugging facilities 407
Figure 145. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 7 of 10)
Figure 145. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 8 of 10)
Figure 145. Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 9 of 10)
Table 53 on page 412 lists the sections of the LEDATA VERBEXIT output, which appear independently of the Language Environment-conforming languages used.
Table 53. Contents of the Language Environment LEDATA VERBEXIT formatted output (AMODE 64)

<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” on page 397.</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>
<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 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 routines, this is the compile unit name. For Language Environment-conforming assemblers, this is the ENTNAME = value on the CELQPRLG 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>• 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>[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 run-time 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 run-time 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>[7] Language Environment Trace Table</td>
<td>If the TRACE run-time option was set to ON, this section shows the contents of the Language Environment trace table.</td>
</tr>
<tr>
<td>Section Number and Heading</td>
<td>Contents</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------</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</td>
<td>These sections are included when the SUMMARY parameter is specified on the LEDATA invocation.</td>
</tr>
</tbody>
</table>
| [9] Summary Header | The summary header section contains:  
• Address of Thread control block (TCB)  
• Release number  
• Address Space ID (ASID) |
| [10] Active Members List | Lists active members, which is extracted from the enclave member list (MEML). |
| [12] CEELCA | 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. |
| [13] CEECAA | 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. |
| [14] CEEPCB | Formats the contents of the Language Environment process control block (PCB), and the process level member list. |
| [16] CEEEDB | Formats the contents of the Language Environment enclave data block (EDB), and the enclave level member list. |
| [17] Run-Time Options | Lists the run-time options in effect at the time of the dump, and indicates where they were set. |
| [18] 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 (ENSQ) and for each different type of heap storage:  
• Heap control block (HPCQ)  
• Chain of heap anchor blocks (HANQ). A HANQ 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 415. |
| [19] Stack Storage Control Blocks | This section is included when the STACK or SM parameter is specified on the LEDATA invocation. This section formats:  
• Stack anchor (SANC)  
• Chain of dynamic save areas (DSA) |
| [20] 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. |
| [21] Message Processing Control Blocks | This section is included when the MH parameter is specified on the LEDATA invocation. |
| [22] Preinitialization Information | This section is included when the PTBL parameter is specified on the LEDATA invocation. It formats information related to preinitialization. See "PTBL LEDATA output" on page 414 for more information. If the preinitialization service CELQPIPI was not used to initialize this environment, the message: No PIPICB associated with CAA is displayed instead. |
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 146 illustrates the output produced when the VERBEXIT LEDATA command is invoked with the PTBL parameter.
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. Figure 147 on page 416 shows the output produced by specifying the HEAP option. "Heap report sections of the LEDATA output" on page 423 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.
HEAP

64 BIT LANGUAGE ENVIRONMENT DATA

Language Environment Product 04 V01 R09.00

Heap Storage Control Blocks

Heap pools trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108
+000000 EYE_CATCHER:ENSQ HEAPALLOC_VAL:00000000
+000014 IPT_TOKEN:00008000 00000002 00000016 007FF030
+000024 HEAPLOCKWORD:00000000 RPT_STOR:00000001_00100410
+000030 UHEAP64:C8D7C3D8 00000000 00000001 001005B0 00000001 001007F0
+000036 LHEAP64:C8D7C3D8 00000000 00000001 001005E0 00000001 00100670
+000060 THNQ: 00000000_00100138
+000000 EYE_CATCHER:THNQ FLAGS:80000000
+000010 NEXT:00000000_001007F0
+000020 PREV:00000000_00100138
+000028 SEGMENT:00000000_08300000
+000030 ROOT:00000000_083CF0C0
+000036 ROOT_LEN:00000000_00030F40

User Heap64 Control Blocks

HPCQ: 00000001_00100138
+000000 EYE_CATCHER:HPCQ FIRST:00000000_001005B0 LAST:00000001_001007F0
+000018 INITSIZE:00000000 00000001 INCRSIZE:00000000 00000001
+00002C OPTIONS:80000000

HPSQ: 00000001_00005058

THNQ: 00000001_001005B0
+000000 EYE_CATCHER:THNQ FLAGS:80000000 NEXT:00000000_001007F0
+000010 PREV:00000000_00100138
+000020 SEGMENT:00000000_08300000

HANQ: 00000001_00300000
+000000 EYE_CATCHER:HANQ FLAGS:80000000 HEAPID:00000001_00100138
+000020 SEGMENT:00000000_08300000

[1] Free Storage Tree for Heap Segment 00000001

Figure 147. Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 1 of 8)
Figure 147. Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 2 of 8)
Free Storage Tree for Heap Segment 00000001

<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>0000000108400000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 00000001

To display entire segment: IP LIST 000000000108400000 LEN(X'0000000001000000') ASID(X'0028')

Free storage element, length=00000000000FFFC0.

To display: IP LIST 000000000108400040 LEN(X'00000000000FFFC0') ASID(X'0028')

Summary of analysis for Heap Segment 0000000108400000:

Amounts of identified storage: Free:000FFFC0 Allocated:00000000 Total:000FFFC0

Number of identified areas : Free: 1 Allocated: 0 Total: 1

00000000 bytes of storage were not accounted for.

No errors were found while processing this heap segment.

THNQ: 00000001_00100610
+E00000 EYE_CATCHER:THNQ FLAGS:00000000 NEXT:00000001_00100640
+E00010 PREV:00000001_001005E0 HEAPID:00000001_00100168
+E00020 SEGMENT:00000001_08800000 SEG_LEN:00000000 00200000

HANQ: 00000001_08800000
+E00000 EYE_CATCHER:HANQ FLAGS:00000000 HEAPID:00000001_00100168
+E00020 SEGMENT:00000001_08800000 ROOT:00000001_08943AC0
+E00030 SEG_LEN:00000000 00200000 ROOT_LEN:00000000 000BC540

Free Storage Tree for Heap Segment 00000001

<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>0000000108943AC0</td>
<td>00000000000BC540</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 00000001

0000000108800000: Allocated storage element, length=0000000000100040.

To display: IP LIST 0000000108800040 LEN(X'0000000000100040') ASID(X'0028')

0000000108900080: Allocated storage element, length=00000000000132A0.

To display: IP LIST 0000000108900080 LEN(X'00000000000132A0') ASID(X'0028')

0000000108943600: Allocated storage element, length=00000000000004C0.

To display: IP LIST 0000000108943600 LEN(X'00000000000004C0') ASID(X'0028')

Summary of analysis for Heap Segment 0000000108800000:

Amounts of identified storage: Free:000BC540 Allocated:00143A80 Total:001FFFC0

Number of identified areas : Free: 1 Allocated: 8 Total: 9

00000000 bytes of storage were not accounted for.

No errors were found while processing this heap segment.

THNQ: 00000001_00100640
+E00000 EYE_CATCHER:THNQ FLAGS:00000000 NEXT:00000001_00100670
+E00010 PREV:00000001_00100610 HEAPID:00000001_00100168
+E00020 SEGMENT:00000001_08A00000 SEG_LEN:00000000 00400000

HANQ: 00000001_08A00000
+E00000 EYE_CATCHER:HANQ FLAGS:00000000 HEAPID:00000001_00100168
+E00020 SEGMENT:00000001_08A00000 ROOT:00000001_08D48340
+E00030 SEG_LEN:00000000 00400000 ROOT_LEN:00000000 000B7CC0

Figure 147. Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 3 of 8)
Free Storage Tree for Heap Segment 00000001

<table>
<thead>
<tr>
<th>Depth</th>
<th>Node</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>0</td>
<td>00000000108D483A0</td>
<td>00000000000B7CC0</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 00000001

To display entire segment: IP LIST 00000000108A00000 LEN(X'0000000000000000') ASID(X'0028')

0000000108A000040: Allocated storage element, length=00000000000B7CC0.
To display: IP LIST 00000000108A000040 LEN(X'00000000000B7CC0') ASID(X'0028')

0000000108A000050: C8D7E3C1 0000000C 000000F0 00000000 00000000 00000008 08A460F8 |HPLA......0........H.....u-8|

0000000108D48340: Free storage element, length=00000000000B7CC0.
To display: IP LIST 00000000108D48340 LEN(X'00000000000B7CC0') ASID(X'0028')

Summary of analysis for Heap Segment 0000000108A00000:
Amounts of identified storage: Free:000B7CC0 Allocated:00348300 Total:003FFFC0
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.
** NO SEGMENTS ALLOCATED **

Library Heap31 Control Blocks

HPCQ: 00000001_001001CB
  +000000 EYE_CATCHER:HPCQ FIRST:00000001_001007C0 LAST:00000001_001007C0
  +000018 INITSIZE:00000000 00004000 INCRSIZE:00000000 00002000
  +00002C OPTIONS:50000000

THNQ: 00000001_001001CB
  +000000 EYE_CATCHER:THNQ FLAGS:00000000 NEXT:00000001_001001CB
  +000010 PREV:00000001 001001CB HEAPID:00000001_001001CB
  +000020 SEGMENT:00000000_25773000 SEG_LEN:00000000 00004000

HANQ: 00000000_25773000
  +000000 EYE_CATCHER:HANQ FLAGS:00000000 HEAPID:00000001_001001CB
  +000020 SEGMENT:25773000 ROOT:25774168 SEG_LEN:00004000
  +00002C ROOT_LEN:00002E98

This is the last heap segment in the current heap.

---

This is the last heap segment in the current heap.

Free Storage Tree for Heap Segment 00000000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25774168</td>
<td>00002E98</td>
</tr>
</tbody>
</table>

Map of Heap Segment 00000000

To display entire segment: IP LIST 00000000 LEN(X'00004000') ASID(X'0028')

Summary of analysis for Heap Segment 00000000:

Amounts of identified storage: Free:00000E98 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 Heap24 Control Blocks

HPCQ: 00000001_001001F8
  +000000 EYE_CATCHER:HPCQ FIRST:00000001_001000F8 LAST:00000001_001001F8
  +000018 INITSIZE:00000000 00001000 INCRSIZE:00000000 00001000
  +00002C OPTIONS:30000000

** NO SEGMENTS ALLOCATED **

Library Heap24 Control Blocks

HPCQ: 00000001_00100228
  +000000 EYE_CATCHER:HPCQ FIRST:00000001_00100028 LAST:00000001_00100228
  +000018 INITSIZE:00000000 00002000 INCRSIZE:00000000 00001000
  +00002C OPTIONS:30000000
Figure 147. Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 6 of 8)
Figure 147. Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 7 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 54. 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>
<tbody>
<tr>
<td>[1] Free Storage Tree Report</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>• Falls on a doubleword boundary</td>
</tr>
<tr>
<td></td>
<td>• Falls within the current heap segment</td>
</tr>
<tr>
<td></td>
<td>• Does not point to itself</td>
</tr>
<tr>
<td></td>
<td>• Does not point to a node that was previously traversed</td>
</tr>
<tr>
<td></td>
<td>Each node length is validated to ensure that it:</td>
</tr>
<tr>
<td></td>
<td>• Is a multiple of 8</td>
</tr>
<tr>
<td></td>
<td>• Is not larger than the heap segment length</td>
</tr>
<tr>
<td></td>
<td>• Does not cause the end of the node to fall outside of the current heap segment</td>
</tr>
<tr>
<td></td>
<td>• Does not cause the node to overlap another node</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

| [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 run-time 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 run-time 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 run-time 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 run-time 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 heappool LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates a detailed
heappool report when heappools is ON. The detailed heappool report is useful
when trying to find potential damaged cells because it provides very specific
information. Figure 148 on page 426 illustrates the details of heappool report.
"Heappool report sections of the LEDATA output" on page 430 describes the
information contained in the formatted output.
Heap Pool Report

QPCB: 00000008_08733600
+000000 EYECATCHER:QPCB LENGTH:00001800 NUMPOLS:00000010
+00000C LARGECELL_SIZE:00000000 BIG_REQUESTS:00000000
+000018 STORAGE_HITS_ADDR:00000000_00000000 FLAGS:0400
+000022 NUMGETARRAYS:05 NUMCELLSIZE:0C
+000028 GET_POOLINFO_ARRAYS_PTR:00000008_08733800

- Data for pool 1:
POOLDATA: 00000008_08733D00
+000000 POOL_INDEX:00000001 INPUT_CELL_SIZE:00000008
+000008 CELL_SIZE:00000020 INPUT_COUNT:0000000A
+000010 CELL_POOL_SIZE:00000140 CELL_POOL_NUM:0000000A
+000018 POOL_LATCH_ADDR:00000008_087117E0 POOL_EXTENTS:00000001
+000028 LAST_CELL:00000008_0862E680 NEXT_CELL:00000008_0862E580
+000040 Q_CONTROL_INFO:00000000 00000005 Q_FIRST_CELL:00000008_0862E560
+000050 POOL_NUM_GET_TOTAL:00000000 00000003
+000058 POOL_NUM_FREE:00000000 00000001 POOL_EXTENTS_ANCHOR:00000008_0862E550
+000068 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:01
+00006A POOL_NUM_SAME_SIZE:01 POOL_TRACE_TABLE:00000008_088000E0

EXTENT: 00000008_0862E550
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_00000000
To display entire pool extent: IP LIST 000000080862E550 LEN('00000150') ASID('0021')
000000080862E560: Free storage cell. To display: IP LIST 000000080862E560 LEN('00000020') ASID('0021')

[1] Verifying free chain for pool: 1...
No errors were found while processing free chain.

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:
POOLDATA: 00000008_08733E00
+000000 POOL_INDEX:00000002 INPUT_CELL_SIZE:00000020
+000008 CELL_SIZE:00000030 INPUT_COUNT:00000004
+000010 CELL_POOL_SIZE:000000C0 CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:00000008_08711808 POOL_EXTENTS:00000001
+000028 LAST_CELL:00000008_0862E750 NEXT_CELL:00000008_0862E6F0
+000040 Q_CONTROL_INFO:00000000 00000010 Q_FIRST_CELL:00000008_0862E6C0
+000050 POOL_NUM_GET_TOTAL:00000000 00000003
+000058 POOL_NUM_FREE:00000000 00000001 POOL_EXTENTS_ANCHOR:00000008_0862E6B0
+000068 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:02
+00006A POOL_NUM_SAME_SIZE:01 POOL_TRACE_TABLE:00000008_08846110

EXTENT: 00000008_0862E6B0
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_00000000
To display entire pool extent: IP LIST 000000080862E6B0 LEN('000000D0') ASID('0021')
000000080862E6C0: Free storage cell. To display: IP LIST 000000080862E6C0 LEN('00000030') ASID('0021')

[1] Verifying free chain for pool: 2...
No errors were found while processing free chain.

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:
POOLDATA: 00000008_08733F00
+000000 POOL_INDEX:00000003 INPUT_CELL_SIZE:00000080
+000008 CELL_SIZE:00000090 INPUT_COUNT:00000004
+000010 CELL_POOL_SIZE:00000240 CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:00000008_08711830 POOL_EXTENTS:00000002
+000028 LAST_CELL:00000008_0862E950 NEXT_CELL:00000008_0862E950
+000040 Q_CONTROL_INFO:00000000 00000015 Q_FIRST_CELL:00000008_0862E830
+000050 POOL_NUM_GET_TOTAL:00000000 0000000E
+000058 POOL_NUM_FREE:00000000 00000002 POOL_EXTENTS_ANCHOR:00000008_0862E790
+000068 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:03
+00006A POOL_NUM_SAME_SIZE:01 POOL_TRACE_TABLE:00000008_0888C140

Figure 148. Example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64) (Part 1 of 5)
Heap Pool Extent Mapping
EXTENT: 00000000_0862E8F0
+000000 EYE_CATCHER: EX64 NEXT_EXTENT: 00000000_0862E2F0
To display entire pool extent: IP LIST 00000000_0862E790 LEN('X'00000250') ASID('X'0021')
00000000_0862E7A0: Free storage cell. To display: IP LIST 00000000_0862E7A0 LEN('X'0000090') ASID('X'0021')
00000000_0862E7B0: Free storage cell. To display: IP LIST 00000000_0862E7B0 LEN('X'0000090') ASID('X'0021')
00000000_0862E7C0: Allocated storage cell. To display: IP LIST 00000000_0862E7C0 LEN('X'0000090') ASID('X'0021')
00000000_0862E7D0: 00000008_0862D2D0 00000000 000005E8 00000000 0000000B 00000000 00000008|......K}.......Y................|
EXTENT: 00000000_0862E2F0
+000000 EYE_CATCHER: EX64 NEXT_EXTENT: 00000000_00000000
To display entire pool extent: IP LIST 00000000_0862E2F0 LEN('X'00000250') ASID('X'0021')
00000000_0862E300: Allocated storage cell. To display: IP LIST 00000000_0862E300 LEN('X'0000090') ASID('X'0021')
00000000_0862E310: C3C4D3D3 00000000 00000000 00000000 C0000000 00000000 00000000 00000000|CDLL............{...............|
00000000_0862E390: Allocated storage cell. To display: IP LIST 00000000_0862E390 LEN('X'0000090') ASID('X'0021')
00000000_0862E3A0: 00000000 00000000 00000008 08C65E90 20004000 00000000 00000000 00000000|.............F;... .............|
00000000_0862E420: Allocated storage cell. To display: IP LIST 00000000_0862E420 LEN('X'0000090') ASID('X'0021')
00000000_0862E430: 00000000 00000000 00000008 00005968 20004000 00000000 00000000 00000000|.................. .............|
EXTENT: 00000000_0862E9F0
+000000 EYE_CATCHER: EX64 NEXT_EXTENT: 00000000_00000000
To display entire pool extent: IP LIST 00000000_0862E9F0 LEN('X'00000450') ASID('X'0021')
00000000_0862EAF0: Free storage cell. To display: IP LIST 00000000_0862EAF0 LEN('X'00000110') ASID('X'0021')
00000000_0862EB10: Free storage cell. To display: IP LIST 00000000_0862EB10 LEN('X'00000110') ASID('X'0021')

Verifying 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 POOLDATA_INDEX: 00000004 INPUT_CELL_SIZE: 000000100
+000008 CELL_POOL_SIZE: 000000110 INPUT_COUNT: 00000004
+000010 CELL_POOL_INDEX: 00000004 CELL_POOL_NUM: 00000004
+000018 POOL_LATCH_ADDR: 00000000_08711858 POOL_EXTENTS: 00000001
+000028 LAST_CELL: 00000000_08605B10 NEXT_CELL: 00000000_08605B20
+000040 Q_CONTROL_INFO: 00000000 00000000 Q_FIRST_CELL: 00000000_0862EB10
+000050 POOL_NUM_GET_TOTAL: 00000000 00000001
+000058 POOL_NUM_FREE: 00000000 00000002 POOL_EXTENTS_ANCHOR: 00000000_0862E9F0
+000068 POOL_INDEX_SAME_SIZE: 01 POOL_INDEX_SIZE: 04
+00006A POOL_NUM_SAME_SIZE: 01 POOL_TRACE_TABLE: 00000000_088D2170

Verifying 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 POOLDATA_INDEX: 00000005 INPUT_CELL_SIZE: 00000040
+000008 CELL_SIZE: 00000040 INPUT_COUNT: 00000004
+000010 CELL_POOL_SIZE: 00000004 CELL_POOL_NUM: 00000004
+000018 POOL_LATCH_ADDR: 00000000_08711858 POOL_EXTENTS: 00000001
+000028 LAST_CELL: 00000008_08605B10 NEXT_CELL: 00000008_08605B20
+000040 Q_CONTROL_INFO: 00000000 00000000 Q_FIRST_CELL: 00000000_0862EB10
+000050 POOL_NUM_GET_TOTAL: 00000000 00000001
+000058 POOL_NUM_FREE: 00000000 00000000 POOL_EXTENTS_ANCHOR: 00000008_0862E9F0
+000068 POOL_INDEX_SAME_SIZE: 01 POOL_INDEX_SIZE: 05
+00006A POOL_NUM_SAME_SIZE: 05 POOL_TRACE_TABLE: 00000000_089181A0

Verifying free chain for pool: 5.1...
No errors were found while processing free chain.
Summary of analysis for Pool 5.1:
Number of cells: Unused: 0 Free: 2 Allocated: 0 Total Used: 2
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Figure 148. Example formatted detailed heappool report from LEDATA VERBEXIT (AMODE 64) (Part 2 of 5)
Heap Pool Extent Mapping
EXTENT: 00000008_086056F0

+000000 EYE_CATCHER:EX64 NEXT EXTENT:00000000 00000000
To display entire pool extent: IP LIST 00000008086056F0 LEN(X'00001050') ASID(X'0021')
0000000808605700: Allocated storage cell. To display: IP LIST 0000000808605700 LEN(X'00000410') ASID(X'0021')
0000000808605710: 00000008 08606770 00000008 08606958 00000008 08606995 00000008 086069D2|.....-.......-.......-.n.....-.K|

Summary of analysis for Pool 5.1:
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.

Data for pool 5.2:
POOLDATA: 00000008_08734200
+000000 POOLDATA:00000006 INPUT_CELL_SIZE:00000400
+000008 CELL_SIZE:00000410 INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00000004 CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:00000008_08711880 POOL_INDEX:00000001
+000020 LAST_CELL:00000008_0862FA90 NEXT_CELL:00000008_0862FA90
+000040 Q_CONTROL_INFO:00000000 00000000 Q_FIRST_CELL:00000008_0862F680
+000050 POOL_NUM_GET_TOTAL:00000000 00000009
+000058 POOL_NUM_FREE:00000000 00000003 POOL_EXTENTS_ANCHOR:00000008_0862EE50
+000068 POOL_INDEX_SAME_SIZE:02 POOL_INDEX_SAME_SIZE:02 POOL_INDEX_SIZE:05
+00006A POOL_INDEX_SIZE:05 POOL_INDEX_SIZE:05 POOL_TRACE_TABLE:00000008_0895E1D0

Verifying free chain for pool: 5.2...
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
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 5.3:
POOLDATA: 00000008_08734300
+000000 POOLDATA:00000006 INPUT_CELL_SIZE:00000400
+000008 CELL_SIZE:00000410 INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00000004 CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:00000008_08711880 POOL_INDEX:00000000
+000020 LAST_CELL:00000008_0862FA90 NEXT_CELL:00000008_0862FA90
+000040 Q_CONTROL_INFO:00000000 00000000 Q_FIRST_CELL:00000008_0862F680
+000050 POOL_NUM_GET_TOTAL:00000000 00000000
+000058 POOL_NUM_FREE:00000000 00000000 POOL_EXTENTS_ANCHOR:00000000_00000000
+000068 POOL_INDEX_SAME_SIZE:03 POOL_INDEX_SAME_SIZE:03 POOL_INDEX_SIZE:05
+00006A POOL_INDEX_SIZE:05 POOL_INDEX_SIZE:05 POOL_TRACE_TABLE:00000008_089A4200
There are no extents for this pool.

Data for pool 5.4:
POOLDATA: 00000008_08734400
+000000 POOLDATA:00000006 INPUT_CELL_SIZE:00000400
+000008 CELL_SIZE:00000410 INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00000004 CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:00000008_08711880 POOL_INDEX:00000000
+000020 LAST_CELL:00000008_00000000 NEXT_CELL:00000008_00000000
+000040 Q_CONTROL_INFO:00000000 00000000 Q_FIRST_CELL:00000008_00000000
+000050 POOL_NUM_GET_TOTAL:00000000 00000000
+000058 POOL_NUM_FREE:00000000 00000000 POOL_EXTENTS_ANCHOR:00000000_00000000
+000068 POOL_INDEX_SAME_SIZE:04 POOL_INDEX_SAME_SIZE:04 POOL_INDEX_SIZE:05
+00006A POOL_INDEX_SIZE:05 POOL_INDEX_SIZE:05 POOL_TRACE_TABLE:00000008_089EA230
There are no extents for this pool.

Figure 148. Example formatted detailed heappool report from LEDATA VERBEXIT (AMODE 64) (Part 3 of 5)
Data for pool 5.5:
POOLDATA: 00000008_08734500
+000000 POOLDATA: 00000008_08734500
+000000 INPUT_CELL_SIZE:00000000 INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000018 POOL_LATCH_ADDR:00000008_08711880 POOL_INDEX:00000009
+000028 LAST_CELL:00000000_00000000 NEXT_CELL:00000000_00000000
+000030 Q_CONTROL_INFO:00000000 00000000 Q_FIRST_CELL:00000000_00000000
+000038 POOL_NUM_GET_TOTAL:00000000 00000000
+000040 POOL_NUM_FREE:00000000 00000000
+000048 POOL_INDEX_SAME_SIZE:05 POOL_INDEX_SIZE:05
+000050 POOL_NUM_SAME_SIZE:05 POOL_TRACE_TABLE:00000008_08A30260
+000058 POOL_TRACE_TABLE:00000008_08A30260
+000060 POOL_TRACE_TABLE:00000008_08A30260
There are no extents for this pool.

Data for pool 6:
POOLDATA: 00000008_08734600
+000000 POOLDATA: 00000008_08734600
+000000 INPUT_CELL_SIZE:00000000 INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000018 POOL_LATCH_ADDR:00000008_087118A8 POOL_INDEX:0000000A
+000028 LAST_CELL:00000000_00000000 NEXT_CELL:00000000_00000000
+000030 Q_CONTROL_INFO:00000000 00000000 Q_FIRST_CELL:00000000_00000000
+000038 POOL_NUM_GET_TOTAL:00000000 00000000
+000040 POOL_NUM_FREE:00000000 00000000
+000048 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:06
+000050 POOL_NUM_SAME_SIZE:01 POOL_TRACE_TABLE:00000008_08A76290
+000058 POOL_TRACE_TABLE:00000008_08A76290
+000060 POOL_TRACE_TABLE:00000008_08A76290
EXTENT: 00000008_0862C290
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_00000000
To display entire pool extent: IP LIST 000000080862C290 LEN(X'00002050') ASID(X'0021')
000000080862D2C0: Allocated storage cell. To display: IP LIST 000000080862D2C0 LEN(X'00000810') ASID(X'0021')
000000080862D2D0: 00000008 082FA5A8 00000000 00000000 00000008 082FAE20 00000000 00000008|......vy........................|
[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 POOLDATA: 00000008_08734700
+000000 INPUT_CELL_SIZE:00000000 INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000018 POOL_LATCH_ADDR:00000008_087118D0 POOL_INDEX:0000000B
+000028 LAST_CELL:00000000_00000000 NEXT_CELL:00000000_00000000
+000030 Q_CONTROL_INFO:00000000 00000000 Q_FIRST_CELL:00000000_00000000
+000038 POOL_NUM_GET_TOTAL:00000000 00000000
+000040 POOL_NUM_FREE:00000000 00000000
+000048 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:07
+000050 POOL_NUM_SAME_SIZE:01 POOL_TRACE_TABLE:00000008_08ABC2C0
+000058 POOL_TRACE_TABLE:00000008_08ABC2C0
EXTENT: 00000008_08606750
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_00000000
To display entire pool extent: IP LIST 0000000808606750 LEN(X'00000250') ASID(X'0021')
0000000808606760: Allocated storage cell. To display: IP LIST 0000000808606760 LEN(X'00000250') ASID(X'0021')
0000000808606770: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000|...xy.........................|
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 POOLDATA: 00000008_08734800
+000000 INPUT_CELL_SIZE:00000000 INPUT_COUNT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000018 POOL_LATCH_ADDR:00000008_087118F8 POOL_INDEX:0000000C
+000028 LAST_CELL:00000000_00000000 NEXT_CELL:00000000_00000000
+000030 Q_CONTROL_INFO:00000000 00000000 Q_FIRST_CELL:00000000_00000000
+000038 POOL_NUM_GET_TOTAL:00000000 00000000
+000040 POOL_NUM_FREE:00000000 00000000
+000048 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:06
+000050 POOL_NUM_SAME_SIZE:01 POOL_TRACE_TABLE:00000008_08ABC2C0
+000058 POOL_TRACE_TABLE:00000008_08ABC2C0
EXTENT: 00000008_08606750
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_00000000
To display entire pool extent: IP LIST 0000000808606750 LEN(X'00000250') ASID(X'0021')
0000000808606760: Allocated storage cell. To display: IP LIST 0000000808606760 LEN(X'00000250') ASID(X'0021')
0000000808606770: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000|...xy.........................|
Summary of analysis for Pool 8:
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.

Figure 148. Example formatted detailed heappool report from LEDATA VERBEXIT (AMODE 64) (Part 4 of 5)

Chapter 12. Using Language Environment AMODE 64 debugging facilities 429
As Table 55 on page 431 shows, the Heappool 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.

Figure 148. Example formatted detailed heappool report from LEDATA VERBEXIT (AMODE 64) (Part 5 of 5)
Table 55. Contents of the Heappool report 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] 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>
<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 the 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 HEAPPOOLS trace report when the HPT option is used (see Figure 149). The argument value is the ID of the pool to be formatted in the report. Table 56 on page 435 explains the contents of each section of the report.

```
HPT(3)
************************************************************
64 BIT LANGUAGE ENVIRONMENT DATA
************************************************************

Language Environment Product 04 V01 R0A.00

[1] HEAPPOLLS64 Trace Table


Type: FREE Cell Address: 00000001086588E0 Cpuid: 01 Tcb: 008D7820

[4] CALL NAME   CALL ADDRESS   CALL OFFSET
GetStorage::~GetStorage() 0000000025B001B0 00000056
foo8() 0000000025B00348 0000006A
foo7() 0000000025B003D8 00000010
foo6() 0000000025B00410 00000010
foo5() 0000000025B00448 00000010
foo4() 0000000025B00480 00000010
foo3() 0000000025B004B8 00000010
foo2() 0000000025B004F0 00000010
foo1() 0000000025B00528 00000000
thread 0000000025B005C8 00000000

Timestamp: 2008/03/14 18:20:40.23975
Type: FREE Cell Address: 0000000108658970 Cpuid: 01 Tcb: 008D7820

CALL NAME   CALL ADDRESS   CALL OFFSET
GetStorage::~GetStorage() 0000000025B001B0 00000056
foo9() 0000000025B00260 0000006E
foo8() 0000000025B00348 00000046
foo7() 0000000025B003D8 00000046
foo6() 0000000025B00410 00000010
foo5() 0000000025B00448 00000010
foo4() 0000000025B00480 00000010
foo3() 0000000025B004B8 00000010
foo2() 0000000025B004F0 00000010
foo1() 0000000025B00528 00000000

Figure 149. Example of formatted detailed HEAPPOOLS trace report from LEDATA VERBEXIT (AMODE 64) (Part 1 of 5)
<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 18:20:40.238021</th>
<th>Type: FREE</th>
<th>Cell Address: 0000000108658970</th>
<th>CpuId: 01</th>
<th>Tcb: 008D7AD0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL NAME</td>
<td>CALL ADDRESS</td>
<td>CALL OFFSET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetStorage::~GetStorage()</td>
<td>0000000025B001B0</td>
<td>00000056</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo8()</td>
<td>0000000025B00348</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo7()</td>
<td>0000000025B003D8</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo6()</td>
<td>0000000025B00410</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo5()</td>
<td>0000000025B00448</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo4()</td>
<td>0000000025B00480</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo3()</td>
<td>0000000025B0048B</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo2()</td>
<td>0000000025B004F0</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo1()</td>
<td>0000000025B0052B</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread</td>
<td>0000000025B005C8</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 149. Example of formatted detailed HEAPPOOLS trace report from LEDATA VERBEXIT (AMODE 64) (Part 2 of 5)
<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 18:20:40.238013</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: GET Cell Address: 00000001086588E0 Cpid: 01 Tcb: 008D7A00</td>
</tr>
<tr>
<td>CALL NAME</td>
</tr>
<tr>
<td>GetStorage::GetStorage(int)</td>
</tr>
<tr>
<td>foo8()</td>
</tr>
<tr>
<td>foo7()</td>
</tr>
<tr>
<td>foo6()</td>
</tr>
<tr>
<td>foo5()</td>
</tr>
<tr>
<td>foo4()</td>
</tr>
<tr>
<td>foo3()</td>
</tr>
<tr>
<td>foo2()</td>
</tr>
<tr>
<td>foo1()</td>
</tr>
<tr>
<td>thread</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 18:20:40.158670</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: GET Cell Address: 0000000108658850 Cpid: 01 Tcb: 008FF038</td>
</tr>
<tr>
<td>CALL NAME</td>
</tr>
<tr>
<td>CEEOPMI</td>
</tr>
<tr>
<td>pthread_create</td>
</tr>
<tr>
<td>main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 18:20:40.140382</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: GET Cell Address: 00000001086587C0 Cpid: 01 Tcb: 008FF038</td>
</tr>
<tr>
<td>CALL NAME</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
</tr>
<tr>
<td>pthread_create</td>
</tr>
<tr>
<td>main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 18:20:40.140373</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: GET Cell Address: 0000000108658730 Cpid: 01 Tcb: 008FF038</td>
</tr>
<tr>
<td>CALL NAME</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
</tr>
<tr>
<td>pthread_create</td>
</tr>
<tr>
<td>main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 18:20:40.023500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: GET Cell Address: 00000001086586A0 Cpid: 01 Tcb: 008FF038</td>
</tr>
<tr>
<td>CALL NAME</td>
</tr>
<tr>
<td>dllinit</td>
</tr>
<tr>
<td>CELQINMN</td>
</tr>
<tr>
<td>CELQINIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 18:20:40.239877</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: FREE Cell Address: 000000002635E058 Cpid: 01 Tcb: 008D7A00</td>
</tr>
<tr>
<td>CALL NAME</td>
</tr>
<tr>
<td>GetStorage::~GetStorage()</td>
</tr>
<tr>
<td>foo8()</td>
</tr>
<tr>
<td>foo7()</td>
</tr>
<tr>
<td>foo6()</td>
</tr>
<tr>
<td>foo5()</td>
</tr>
<tr>
<td>foo4()</td>
</tr>
<tr>
<td>foo3()</td>
</tr>
<tr>
<td>foo2()</td>
</tr>
<tr>
<td>foo1()</td>
</tr>
<tr>
<td>thread</td>
</tr>
</tbody>
</table>

HEAPPOOLS Trace Table

POOLID: 3 ASID: 001F AVAILABLE ENTRIES: 8 OF 8

<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 18:20:40.140382</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: GET Cell Address: 0000000108658730 Cpid: 01 Tcb: 008FF038</td>
</tr>
<tr>
<td>CALL NAME</td>
</tr>
<tr>
<td>thread</td>
</tr>
</tbody>
</table>

Figure 149. Example of formatted detailed HEAPPOOLS trace report from LEDATA VERBEXIT (AMODE 64) (Part 3 of 5)
<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Type</th>
<th>Caller Address</th>
<th>Cpuid</th>
<th>Tcb</th>
<th>CALL NAME</th>
<th>CALL ADDRESS</th>
<th>CALL OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008/03/14 18:20:40.239874</td>
<td>FREE</td>
<td>000000002635E0E0</td>
<td>01</td>
<td>0807B20</td>
<td>GetStorage::GetStorage()</td>
<td>0000000025800180</td>
<td>0000003A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo9()</td>
<td>0000000025800260</td>
<td>0000006E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo8()</td>
<td>0000000025800348</td>
<td>00000046</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo7()</td>
<td>00000000258003D8</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo6()</td>
<td>0000000025800410</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo5()</td>
<td>0000000025800448</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo4()</td>
<td>0000000025800480</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo3()</td>
<td>0000000025800488</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo2()</td>
<td>00000000258004F0</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo1()</td>
<td>0000000025800528</td>
<td>00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Type</th>
<th>Caller Address</th>
<th>Cpuid</th>
<th>Tcb</th>
<th>CALL NAME</th>
<th>CALL ADDRESS</th>
<th>CALL OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008/03/14 18:20:40.239872</td>
<td>GET</td>
<td>000000002635E0E0</td>
<td>01</td>
<td>0807B20</td>
<td>GetStorage::GetStorage()</td>
<td>000000002580011B</td>
<td>0000002E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo9()</td>
<td>0000000025800260</td>
<td>0000003A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo8()</td>
<td>0000000025800348</td>
<td>00000046</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo7()</td>
<td>00000000258003D8</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo6()</td>
<td>0000000025800410</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo5()</td>
<td>0000000025800448</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo4()</td>
<td>0000000025800480</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo3()</td>
<td>0000000025800488</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo2()</td>
<td>00000000258004F0</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo1()</td>
<td>0000000025800528</td>
<td>00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Type</th>
<th>Caller Address</th>
<th>Cpuid</th>
<th>Tcb</th>
<th>CALL NAME</th>
<th>CALL ADDRESS</th>
<th>CALL OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008/03/14 18:20:40.239869</td>
<td>GET</td>
<td>000000002635E058</td>
<td>01</td>
<td>0807B20</td>
<td>GetStorage::GetStorage()</td>
<td>000000002580011B</td>
<td>0000002E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo9()</td>
<td>0000000025800260</td>
<td>0000003A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo8()</td>
<td>0000000025800348</td>
<td>00000046</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo7()</td>
<td>00000000258003D8</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo6()</td>
<td>0000000025800410</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo5()</td>
<td>0000000025800448</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo4()</td>
<td>0000000025800480</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo3()</td>
<td>0000000025800488</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo2()</td>
<td>00000000258004F0</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo1()</td>
<td>0000000025800528</td>
<td>00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Type</th>
<th>Caller Address</th>
<th>Cpuid</th>
<th>Tcb</th>
<th>CALL NAME</th>
<th>CALL ADDRESS</th>
<th>CALL OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008/03/14 18:20:40.238023</td>
<td>FREE</td>
<td>000000002635E058</td>
<td>01</td>
<td>0807A0D</td>
<td>GetStorage::GetStorage()</td>
<td>0000000025800180</td>
<td>0000003A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo9()</td>
<td>0000000025800260</td>
<td>0000006E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo8()</td>
<td>0000000025800348</td>
<td>00000046</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo7()</td>
<td>00000000258003D8</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo6()</td>
<td>0000000025800410</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo5()</td>
<td>0000000025800448</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo4()</td>
<td>0000000025800480</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo3()</td>
<td>0000000025800488</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo2()</td>
<td>00000000258004F0</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo1()</td>
<td>0000000025800528</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>thread</td>
<td>00000000258005C8</td>
<td>00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Type</th>
<th>Caller Address</th>
<th>Cpuid</th>
<th>Tcb</th>
<th>CALL NAME</th>
<th>CALL ADDRESS</th>
<th>CALL OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008/03/14 18:20:40.238018</td>
<td>FREE</td>
<td>000000002635E058</td>
<td>01</td>
<td>0807A0D</td>
<td>GetStorage::GetStorage()</td>
<td>0000000025800180</td>
<td>0000003A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo9()</td>
<td>0000000025800260</td>
<td>0000006E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo8()</td>
<td>0000000025800348</td>
<td>00000046</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo7()</td>
<td>00000000258003D8</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo6()</td>
<td>0000000025800410</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo5()</td>
<td>0000000025800448</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo4()</td>
<td>0000000025800480</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo3()</td>
<td>0000000025800488</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo2()</td>
<td>00000000258004F0</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>foo1()</td>
<td>0000000025800528</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Figure 149. Example of formatted detailed HEAPPOOLS trace report from LEDATA VERBEXIT (AMODE 64) (Part 4 of 5)
Table 56. Contents of a detailed HEAPPOOLS trace report from LEDATA VERBEXIT (AMODE 64)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<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 run-time 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 run-time 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 150 on page 436 illustrates the C/C++-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) run-time option when running the program CELQSAMP. Figure 142 on page 380 describes the C/C++-specific sections of the LEDATA 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 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 10)
Figure 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 2 of 10)
Figure 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 3 of 10)
Figure 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 4 of 10)
Figure 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 5 of 10)
Figure 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 6 of 10)
Figure 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 7 of 10)
Figure 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 8 of 10)
Figure 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 9 of 10)
Table 57 describes the contents of the LEDATA output that is specific to C/C++.

<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>

Figure 150. Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 10 of 10)
Table 57. 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>DCCB</td>
<td>The data control block; for more information, see z/OS DFSMS Macro Instructions for Data Sets.</td>
</tr>
<tr>
<td>DCBHE</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 z/OS Internet Library at <a href="http://www.ibm.com/systems/z/os/zos/bkserv/">http://www.ibm.com/systems/z/os/zos/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>

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 151 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.
Chapter 12. Using Language Environment AMODE 64 debugging facilities 447

Figure 151. Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 4)
Figure 151. Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 2 of 4)
Figure 151. Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 3 of 4)
Figure 151. Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 4 of 4)
Sections of the AUTH LEDATA VERBEXIT formatted output

Table 58 describes the contents of the AUTH LEDATA VERBEXIT formatted output.

Table 58. 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]-[5] Control Blocks for one user managed environment</td>
<td>These sections are 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.</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]-[11] Control Blocks for one set of system managed environments that was initialized</td>
<td>These sections are repeated for each set of system managed environments that was initialized.</td>
</tr>
<tr>
<td>[7] ALES</td>
<td>Each ALES represents a set of system managed environments.</td>
</tr>
<tr>
<td>[8]-[11] Control blocks for one environment definition entry</td>
<td>These sections are repeated for every environment definition entry (AEDE) that was specified when the set of system managed environments 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.</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 above. Each ALRI appears in the table twice, once for the routine name and once for the routine address.</td>
</tr>
<tr>
<td>[10]-[11] Control blocks for one system managed environment</td>
<td>These sections are repeated for every environment associated with the ETINDEX and ALESETE.</td>
</tr>
<tr>
<td>[10] ALEI</td>
<td>Each ALEI control block represents one environment. The ALEIs in this section represent system managed environments.</td>
</tr>
<tr>
<td>[11] ALRI</td>
<td>Contains the ALRI control blocks for each routine that was called in the environment identified by the ALEI. This section does not appear if the environment has not been used to call a routine.</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 MV3 IPCS Commands, SA22-7594.
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 59](#). 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 152](#).

```
CBF 100007B18 struct(CECAA)
```

![Figure 152. CAA formatted by the CBFFORMAT IPCS command](#)

<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>CELDSATR</td>
<td>XPLINK Transition Area</td>
</tr>
</tbody>
</table>
Table 59. 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>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>CELHCOM</td>
<td>CEL Exception Manager Communications Area</td>
</tr>
<tr>
<td>CELHPCQ</td>
<td>Thread Level Heap Control Block</td>
</tr>
<tr>
<td>CELLAA</td>
<td>Library Anchor Area</td>
</tr>
<tr>
<td>CELLCA</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>

Table 60. 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 run-time option is set to ON and:

- A thread ends abnormally because of an unhandled condition of severity 2 or greater and the TERMTHDACT run-time option is set to DUMP, UADUMP, TRACE, or UATRACE.
- An application terminates normally and the TRACE run-time option is set to DUMP (the default).

For more information about recording done by the TERMTHDACT run-time option or the TRACE run-time option, see [z/OS Language Environment Programming](#).

The TRACE run-time option activates Language Environment run-time 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 run-time option. Table 61 summarizes the dump contents that are generated under abnormal termination.

Table 61. TERMTHDACT run-time 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>
<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>
<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. 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.</td>
</tr>
</tbody>
</table>

Under normal termination, with the TRACE run-time 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 run-time 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 153

```
<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 153. 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:
**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 run-time 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 run-time 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 run-time 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 62 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 485.

<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 63 on page 457 shows the Language Environment records that may be generated.
Table 63. 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>
<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 decrement (recursive)</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>
</tbody>
</table>
Table 63. 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>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>
<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>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>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>
<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>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>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>
<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>
</tbody>
</table>
Table 63. 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>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>000008F8</td>
<td>RE</td>
<td>EUI</td>
<td>Error - Uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000008F9</td>
<td>RE</td>
<td>EUO</td>
<td>Error - Unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000008FA</td>
<td>RE</td>
<td>EO</td>
<td>Error - Already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008FB</td>
<td>RE</td>
<td>EIN</td>
<td>Error - Initialization failed (duplicate)</td>
</tr>
</tbody>
</table>

Table 64 shows the format for the Mutex – Condition Variable – Latch entries in the trace table.

Table 64. Format of the mutex/CV/latch records (AMODE 64)

<table>
<thead>
<tr>
<th>Class</th>
<th>Source</th>
<th>Event</th>
<th>Object Addr</th>
<th>Name1</th>
<th>Name2</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 63 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 65 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 485.

Table 65. 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>000000005</td>
<td>Storage allocation entry</td>
</tr>
<tr>
<td>03</td>
<td>000000006</td>
<td>Storage allocation exit</td>
</tr>
</tbody>
</table>
Sample dump for the trace table entry

Figure 154 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>Time 22.02.30.398959</th>
<th>Date 2004.04.08</th>
<th>Thread ID... 2548146000000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member ID.... 03</td>
<td>Flags...... 000000</td>
<td>Entry Type..... 00000005</td>
</tr>
<tr>
<td>94818995 40404040 40404040 40404040</td>
<td>0000000000000001</td>
<td>main 006F printf()</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 in order 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++ run-time library functions, see [z/OS XL C/C++ Run-Time 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` run-time 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 run-time options, TRAP(ON, NOSPIE) and TERMTHDACT(UAIMM). Setting these run-time 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 155 on page 465 shows the structure as it appears in stdio.h.
typedef struct __amrctype {
    union {
        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;
        } __alloc;
        } __code;
    } __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;
    #if __EDC_TARGET >= 0x22080000
    unsigned char __rplfdbwd[4]; /* rpl feedback word */
    #endif /* __EDC_TARGET >= 0x22080000 */
    #if __EDC_TARGET >= 0x41080000
    #ifdef __LP64
    unsigned long __XRBA; /* 8 byte RBA */
    #elif defined(__LL)
    unsigned long long __XRBA; /* 8 byte RBA */
    #else
    unsigned int __XRBA1; /* high half of 8 byte RBA */
    unsigned int __XRBA2; /* low half of 8 byte RBA */
    #endif
    #endif /* QSAM to BSAM switch reason */
    unsigned char __amrc_noseek_to_seek;
    char __amrc_pad[23];
} __amrc_type;

Figure 155. __amrc structure (AMODE 64)

Figure 156 shows the __amrc2 structure as it appears in stdio.h.

struct {
    int __error2;
    char __pad__error2[4];
    FILE *__fileptr;
    int __reserved[6];
}

Figure 156. __amrc2 structure (AMODE 64)

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. 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 66 on page 467.

[8] __msg
May contain the system error messages from read or write operations
emitted from the DFSMS/MVS SYNADF 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 P0INT macros requested (seek) by the XL C/C++ Run-Time
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>Macro</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------</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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>was not specified at allocation</td>
</tr>
</tbody>
</table>

[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 66 lists __last_op values you could receive and where to look for further information.

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__IO_INIT</td>
<td>Will never be seen by SIGIOERR exit value given at initialization.</td>
</tr>
<tr>
<td>__BSAM_OPEN</td>
<td>Sets __error with return code from OS OPEN macro.</td>
</tr>
<tr>
<td>__BSAM_CLOSE</td>
<td>Sets __error with return code from OS CLOSE macro.</td>
</tr>
<tr>
<td>__BSAM_READ</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 66) filled in).</td>
</tr>
<tr>
<td>__BSAM_NOTE</td>
<td>NOTE returned 0 unexpectedly, no return code.</td>
</tr>
<tr>
<td>__BSAM_POINT</td>
<td>This will not appear as an error lastop.</td>
</tr>
<tr>
<td>__BSAM_WRITE</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 65) filled in).</td>
</tr>
<tr>
<td>__BSAM_CLOSE_T</td>
<td>Sets __error with return code from OS CLOSE TYPE=T.</td>
</tr>
<tr>
<td>__BSAM_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 66. __last_op values and diagnosis information (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<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 with no newline such that the record fills up to capacity and subsequent characters cannot be written. For a record I/O file this refers to an fwrite() writing more data than the record can hold. Truncation is always rightmost data. There is no return code.</td>
</tr>
<tr>
<td>__C_FCBCHECK</td>
<td>Set when C or C++ FCB is corrupted. This is due to a pointer corruption somewhere. File cannot be used after this.</td>
</tr>
<tr>
<td>__C_DBCS_TRUNCATE</td>
<td>This occurs when writing DBCS data to a text file and there is no room left in a physical record for anymore double byte characters. A new-line is not acceptable at this point. Truncation will continue to occur until an SI is written or the file position is moved. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_SO_TRUNCATE</td>
<td>This occurs when there is not enough room in a record to start any DBCS string or else when a redundant SO is written to the file before an SI. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_SI_TRUNCATE</td>
<td>This occurs only when there was not enough room to start a DBCS string and data was written anyways, with an SI to end it. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_UNEVEN</td>
<td>This occurs when an SI is written before the last double byte character is completed, thereby forcing C or C++ to fill in the last byte of the DBCS string with a padding byte X'FE'. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_CANNOT_EXTEND</td>
<td>This occurs when an attempt is made to extend a file that allows writing, but cannot be extended. Typically this is a member of a partitioned data set being opened for update.</td>
</tr>
<tr>
<td>__VSAM_OPEN_FAIL</td>
<td>Set when a low level VSAM OPEN fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_RRDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
</tbody>
</table>
Table 66. __last_op values and diagnosis information (AMODE 64) (continued)

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

**Using 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](z/OS XL C/C++ Programming Guide) 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). 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 157 is an example of a routine using perror().

```c
#include <stdio.h>
int main(void){
    FILE *fp;
    fp = fopen("myfile.dat", "w");
    if (fp == NULL)
        perror("fopen error");
}
```

Figure 157. 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++ run-time 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 158 is an example of a routine using __errno2() and Figure 159 on page 472 shows the sample output from that routine.

```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 158. Example of a routine using __errno2() (AMODE 64)
Figure 160 is an example of a routine using the environment variable _EDC_ADD_ERRNO2. Figure 161 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 160. Example of a routine using _EDC_ADD_ERRNO2 (AMODE 64)

fopen() failed: EDC5129I No such file or directory.

Figure 161. Sample output of routine using _EDC_ADD_ERRNO2 (AMODE 64)

fopen() failed: EDC5129I No such file or directory.

Figure 162 on page 473 is an example of a routine using __err2ad() in combination with __errno2(). Figure 163 on page 473 shows the sample output from that routine. For more information about __errno2() and __err2ad(), see z/OS XL C/C++ Run-Time 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 cdivzero shown in Figure 60 on page 221.

---

```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 */
    *__err2ad() = 0x0;
    printf("errno2 = %08x
", __errno2());
    f = fopen(*testfile.dat", "r");
    if (fp == NULL) {
        perror("fopen() failed");
        printf(*errno2 = %08x
", __errno2());
    }
    return 0;
}
```

Figure 162. 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 163. 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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>5823-0:10</th>
<th>Class = automatic,</th>
<th>Location = 2248(r4),</th>
<th>Length = 4</th>
</tr>
</thead>
</table>

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:

<table>
<thead>
<tr>
<th>DSA</th>
<th>Entry</th>
<th>E Offset</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000004</td>
<td>main</td>
<td>+000000016</td>
<td>CDIVZERO</td>
<td></td>
<td></td>
<td>Call</td>
</tr>
</tbody>
</table>

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 cdizero 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>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>parameter</td>
</tr>
<tr>
<td>Location</td>
<td>2432(r4)</td>
</tr>
<tr>
<td>Length</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:

**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 164 shows an example of an aggregate.

```c
typedef struct {
  int asid;
  void *addr;
  asfAmodeType amode;
} asfTargetRef;

asfTargetRef tempTargetRef;
```

Figure 164. Example code for structure variables (AMODE 64)

Figure 165 shows an example of aggregate map.

```
<table>
<thead>
<tr>
<th>Aggregate map for: struct with no tag #68 Total size: 24 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asfTargetRef</td>
</tr>
<tr>
<td>----------------------------------------------------------------</td>
</tr>
<tr>
<td>Offset</td>
</tr>
<tr>
<td>Bytes(Bits)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
</tbody>
</table>
```

Figure 165. 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(+4), 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'1082FDCB8'(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 165. The offset is 8. Add the offset from the Aggregate Map to the address of the `tempTargetRef` variable.

   The result is X'1082FDCC0'(X'1082FDCB8' + 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++ Run-Time 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 166 on page 478 shows a sample C routine that uses the cdump function to generate a dump. Figure 171 on page 481 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 166. Example C routine using cdump() to generate a dump (AMODE 64) (Part 1 of 2)
Figure 166. Example C routine using cdump() to generate a dump (AMODE 64) (Part 2 of 2)

```c
#include <ctest.h>
#pragma linkage(func1, fetchable)
int func1(void) {
    __cdump("This is a sample dump");
    return(0);
}
```

Figure 167. Fetched module for C routine (AMODE 64)

```c
#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 167. Fetched module for C routine (AMODE 64)

**Sample C++ routine that generates a Language Environment dump**

```c
#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 168. Example C++ routine with protection exception generating a dump (AMODE 64)

```c
#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 169 on page 480 shows the template file stack.c
Sample Language Environment dump with C/C++-specific information

This sample dump was produced by compiling the routines shown in Figure 166 on page 478 and Figure 167 on page 479. 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 167 on page 479. 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 171. Example dump from sample C routine (AMODE 64) (Part 1 of 4)
Figure 171. Example dump from sample C routine (AMODE 64) (Part 2 of 4)
Control Blocks Associated with the Thread:

```
CAA(0000000100007AC0)
+0000 0000000100007AC0 00000000 00000000 00000000 00000000 | same as above |
+0010 0000000100007AC0 00000000 00000000 00000000 00000000 | same as above |
+0020 0000000100007AC0 00000000 00000000 00000000 00000000 | same as above |
+0030 0000000100007AC0 00000000 00000000 00000000 00000000 | same as above |
+0040 0000000100007AC0 00000000 00000000 00000000 00000000 | same as above |
```

Math Parameters:

```
+0000 0000000000000000 00000000 00000000 00000000 00000000 | same as above |
+0010 0000000000000000 00000000 00000000 00000000 00000000 | same as above |
```

Thread Synchronization Queue Element (SQEL)(0000000025D820A0)

```
+0000 0000000000000000 00000000 00000000 00000000 00000000 | same as above |
+0010 0000000000000000 00000000 00000000 00000000 00000000 | same as above |
```

Figure 171. Example dump from sample C routine (AMODE 64) (Part 3 of 4)
Figure 171. 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=xxxxxxxx ERRNO2=xxxxxxxx
```

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 run-time library definition side-deck, SCEELIB dataset member CELQS003, on the IMPORT statement for that function.

Figure 172 on page 486 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 455.
Language Environment Trace 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 21.58.20.255215 Date 2004.04.20 Thread ID... 8000000000000000</td>
<td>filebuf::overflow(int)</td>
</tr>
<tr>
<td>+000100</td>
<td>Time 21.58.20.255216 Date 2004.04.20 Thread ID... 8000000000000000</td>
<td>filebuf::overflow(int)</td>
</tr>
<tr>
<td>+000200</td>
<td>Time 21.58.20.255218 Date 2004.04.20 Thread ID... 8000000000000000</td>
<td>filebuf::overflow(int)</td>
</tr>
<tr>
<td>+000300</td>
<td>Time 21.58.20.255243 Date 2004.04.20 Thread ID... 8000000000000000</td>
<td>filebuf::overflow(int)</td>
</tr>
<tr>
<td>+000400</td>
<td>Time 21.58.20.255245 Date 2004.04.20 Thread ID... 8000000000000000</td>
<td>filebuf::overflow(int)</td>
</tr>
</tbody>
</table>

Figure 172. Trace table with XPLINK trace table entries 5 and 6 (AMODE 64) (Part 1 of 2)
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

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 172. Trace table with XPLINK trace table entries 5 and 6 (AMODE 64) (Part 2 of 2)
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 Run-Time 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 174 on page 489](#) shows the generated traceback from the dump.

```c
/* 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 173. C routine with a divide-by-zero error (AMODE 64)*
Information for enclave main

Information for thread 25AC70A000000000

Traceback:

DSA  Entry  E  Offset  Statement  Load Mod  Program Unit  Service Status
00000001  CEEHDSP  +00000000  CELQLIB  CEEHDSP  HLE7730  Cal1
00000002  CELQSIGJ  +0000094E  CELQLIB  CELQSIGJ  HLE7730  Cal1
00000003  CELQHROD  +0000024E  CELQLIB  CELQHROD  HLE7730  Cal1
00000004  CELQHROD  +0000024E  CELQLIB  CELQHROD  HLE7730  Cal1
00000005  CELQHROD  +0000024E  CELQLIB  CELQHROD  HLE7730  Cal1
00000006  funcb  +00000052  CDIVZERO  CDIVZERO  Exception
00000007  main  +00000026  CDIVZERO  CDIVZERO  Call
00000008  CELQINIT  +00001340  CELQLIB  CELQINIT  HLE7730  Call

DSA  DSA Addr  E Addr  PU Addr  PU Offset  Comp Date  Attributes
00000001  00000001082FAAC0  0000000025793E88  0000000025793E88  00000000  20060109  XPLINK  EBCDIC  POSIX  Floating Point
00000002  00000001082FD3E0  000000002588FE28  000000002588FE28  0000094E  20051214  XPLINK  EBCDIC  POSIX  Floating Point
00000003  00000001082FDDE0  00000000257B50D8  00000000257B50D8  0000024E  20051214  XPLINK  EBCDIC  POSIX  Floating Point
00000004  00000001082FE020  00000000258894E0  00000000258894E0  1D58AE60  20051214  XPLINK  EBCDIC  POSIX  Floating Point
00000005  00000001082FE40  00000000257B50D8  00000000257B50D8  0000024E  20051214  XPLINK  EBCDIC  POSIX  Floating Point
00000006  00000001082FF080  00000000257080D0  0000000000000000  ********  20060221  XPLINK  EBCDIC  POSIX  IEEE
00000007  00000001082FF180  0000000025708170  0000000000000000  ********  20060221  XPLINK  EBCDIC  POSIX  IEEE
00000008  00000001082FF280  000000002570B010  000000002570B010  00001340  20051214  XPLINK  EBCDIC  POSIX  Floating Point

Fully Qualified Names

DSA  Entry Program Unit Load Module
00000006  funcb  PLPSC://POSIX.CRTL.C(CDIVZERO)' /u/alfcar/tools/CDIVZERO
00000007  main  PLPSC://POSIX.CRTL.C(CDIVZERO)' /u/alfcar/tools/CDIVZERO

Condition Information for Active Routines

Condition Information for PLPSC://POSIX.CRTL.C(CDIVZERO)' (DSA address 00000001082FF080)

CIB Address: 00000001082FBE00
Current Condition:
CEE0198S The termination of a thread was signaled due to an unhandled condition.
Original Condition:
CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).
Location:
Program Unit: PLPSC://POSIX.CRTL.C(CDIVZERO)'
Entry: funcb Statement: 18 Offset: +00000052
Machine State:
ILC..... 0002 Interruption Code..... 0009
PSW..... 0785240180000000 0000000025708124
GPR0..... **************** GPR1..... **************** GPR2..... **************** GPR3..... ****************
GPR4..... 00000001082FF080 GPR5..... BBBBBBBBBBBBBBB GPR6..... 0000000108414340 GPR7..... 0000000108413430
GPR8..... 0000000025700CC8 GPR9..... 00000000257081B8 GPR10.... 0000000025708278 GPR11.... 0000000108414330
GPR12.... 4040404040404040 GPR13.... 4040404040404040 GPR14.... 4040404040404040 GPR15.... 4040404040404040
FPC...... 00000000
FPR0..... 4327CCCA 93BCEEF1 FPR1..... 00000000 00000000
FPR2..... 00000000 00000000 FPR3..... 00000000 00000000
FPR4..... 3C100000 00000000 FPR5..... 00000000 00000000
FPR6..... 34100000 00000000 FPR7..... 00000000 00000000
FPR8..... 00000000 00000000 FPR9..... 00000000 00000000
FPR10.... 00000000 00000000 FPR11.... 00000000 00000000
FPR12.... 00000000 00000000 FPR13.... 00000000 00000000
FPR14.... 00000000 00000000 FPR15.... 00000000 00000000
Storage dump near condition, beginning at location(0000000025708112):
+0000 0000000025708112 004E300 70000014 A70BFFB7 8E600020 |..T.....x....-..|
+0010 0000000025708122 1D60B904 00075000 48C0E320 48C00014 |.-....&...T.....|

Figure 174. 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:
\[ aa = \text{funcb}(&aa); \]
in the listing. This is where entry \text{funcb} is called. Similarly, statement number 18, corresponding to DSA 6, points to line:
\[ \text{result} = \frac{fa}{(\text{statint}-73)}; \]
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 175 on page 491.

The offset (within \text{funcb}) of the exception from the traceback (X'52') reveals the divide instruction: \text{DR R6,R0} at that location. Instructions at offsets X'32' through X'58' refer to the line:
\[ \text{result} = \frac{fa}{(\text{statint}-73)}; \]
line of the C/C++ routine.
Figure 175. Pseudo assembly listing (AMODE 64) (Part 1 of 3)
### Pseudo assembly listing (AMODE 64) (Part 2 of 3)

```assembly
000001 | * / * C Routine with a Divide-by-Zero Error from LE Debugging Guide */
000002 | * #pragma options(noinline)
000003 | * #include <stdio.h>
000004 | * #include <stdlib.h>
000005 | * #include <errno.h>
000006 | * int statint = 73;
000007 | * int fa;
000008 | * int funcb(int mp);
000009 | * int main(void) {

000010 | * int aa, bb=1;
000011 | * aa = bb;
000012 | * aa = funcb(a);
000013 | * return(aa);
000014 | * }

000015 | }

Constant Area

000080 0985A2A4 93A3407E 406C8415 00 [Result = %d..]

PPA1: Entry Point Constants

```

Figure 175. Pseudo assembly listing (AMODE 64) (Part 2 of 3)
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 176 shows the WSA address.

Enclave Control Blocks:

<table>
<thead>
<tr>
<th>WSA Addr</th>
<th>Module Addr</th>
<th>Thread ID</th>
<th>Use Count</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000108300050</td>
<td>00000001 main</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 176. 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 177 on page 494. 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.

---

CLASS C_WSA64
LENGTH = 38
ATTRIBUTES = MRG, DEFER, AMODE=64
OFFSET = 0 IN SEGMENT 002
ALIGN = QWORD

---

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OFFSET</th>
<th>NAME</th>
<th>TYPE</th>
<th>LENGTH</th>
<th>SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>$PRIV000012</td>
<td>PART</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>CDIVZERO#S</td>
<td>PART</td>
<td>20</td>
<td>CDIVZERO#C</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>statint</td>
<td>PART</td>
<td>4</td>
<td>statint</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>fa</td>
<td>PART</td>
<td>4</td>
<td>fa</td>
<td></td>
</tr>
</tbody>
</table>

---

Figure 177. Writable static map (AMODE 64)

Calling a nonexistent function

Figure 179 on page 495 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 180 on page 496. 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 Run-Time 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 179. 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 Run-Time Messages.

---

/* C/C++ Example of Calling a Nonexistent Subroutine */
/* from LE Debugging Guide */
#pragma options(noinline)
#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 179. C/C++ example of calling a nonexistent subroutine (AMODE 64)
Information for thread 8000000000000000

Traceback:

<table>
<thead>
<tr>
<th>DSA Entry</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CELQHSP</td>
<td>+00000000</td>
<td>CELQLIB</td>
<td>CELQHSP</td>
<td>D1908 Call</td>
</tr>
<tr>
<td>2</td>
<td>CELQHROD</td>
<td>+00000024E</td>
<td>CELQLIB</td>
<td>CELQHROD</td>
<td>D1908 Call</td>
</tr>
<tr>
<td>3</td>
<td>funca</td>
<td>-20900000</td>
<td>EXIST</td>
<td>EXIST</td>
<td>Call</td>
</tr>
<tr>
<td>4</td>
<td>main</td>
<td>+00000034</td>
<td>12</td>
<td>EXIST</td>
<td>Call</td>
</tr>
<tr>
<td>5</td>
<td>CELQINIT</td>
<td>+0000013C</td>
<td>CELQLIB</td>
<td>CELQINIT</td>
<td>D1908 Call</td>
</tr>
</tbody>
</table>

DSA DSA Addr  E Addr  PU Addr  PU Offset  Comp Date  Compile Attributes
1 00000001082FC520 0000000020AB3680 0000000020AB3680 00000000 20061215 CEL XPLINK EBCDIC HFP
2 00000001082FEE40 0000000020AC6A00 0000000020AC6A00 000024E 20061215 CEL XPLINK EBCDIC HFP
3 00000001082FF080 0000000020900000 0000000000000000 ******** 20061215 CEL XPLINK EBCDIC HFP
4 00000001082FF180 0000000020900138 0000000000000000 ******** 20061215 CEL XPLINK EBCDIC HFP
5 00000001082FF280 0000000020903010 0000000020903010 0000134C 20061215 CEL XPLINK EBCDIC HFP

Condition Information for Active Routines

Condition Information for (DSA address 00000001082FF080)
CIB Address: 00000001082FD860
Current Condition:
CEE3985 The termination of a thread was signaled due to an unhandled condition.
Original Condition:
CEE3201S The system detected an operation exception (System Completion Code=0C1).
Location:
Program Unit: Entry: funca Statement: Offset: -20900000
Possible Bad Branch: Statement: 17 Offset: +00000036
Machine State:
ILC..... 0002 Interruption Code..... 0001
PSW..... 07B5201000000000 0000000000000000
GPR0..... 0000000108300060 GPR1..... 00000001082FFA40 GPR2..... 0000000108401F60 GPR3..... 0000000108400070 GPR4..... 00000001082FFA40 GPR5..... 0000000130E1 GPR6..... 0000000000000000 GPR7..... 00000000209002A8 GPR8..... 00000000209000DC GPR9..... 00000000209001A0 GPR10.... 00000000209002A8 GPR11.... 0000000108401F50 GPR12.... 0000000100005340 GPR13.... 000000006F58 GPR14.... 0000000020B4E0A0 GPR15.... 000000000000001F

Storage dump near condition, beginning at location(0000000000000000)
+0000 0000000000000000 Inaccessible storage.
+0010 0000000000000010 Inaccessible storage.
+0020 0000000000000001 Inaccessible storage.

GPREG STORAGE:

Storage around GPR0 (0000000108300060)
-0020 0000000108300040 0000000108300040 0000000000000000
+0030 0000000108300090 0000000108300090 0000000000000000

Storage around GPR1 (00000001082FFA40)
-0020 00000001082FFA20 00000001082FFA20 00000001082FFA20
-0010 00000001082FFA20 - FF7FF 00000001082FFA2F
+0000 00000001082FFA40 00000001082FFA40 00000001082FFA40
+0010 00000001082FFA50 00000001082FFA50 00000001082FFA5F
+0020 00000001082FFA60 - +0003F 00000001082FFA7F

Figure 180. Sections of the dump from example C routine (AMODE 64) (Part 1 of 3)
Figure 180. Sections of the dump from example C routine (AMODE 64) (Part 2 of 3)
2. Find the branch instructions for funca in the listing in Figure 181 on page 499. 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 179 on page 495.
/* C/C++ Example of Calling a Nonexistent Subroutine */

#pragma options(noinline)

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

void funca(int* aa)
{
  int (*func_ptr)(void)=0;

  int main(void) {
    *aa = func_ptr();
  }
}

*** General purpose registers used: 1111111110000000
*** Floating point registers used: 1111111100000000
*** Size of register spill area: 256(max) 0(used)
*** Size of dynamic storage: 0
*** Size of executable code: 86

Figure 181. Pseudo assembly listing (AMODE 64) (Part 1 of 2)
3. Find the offset of func_ptr in the Writable Static Map, shown in Figure 182.

4. Add the offset of func_ptr (X'40') to the address of WSA (X'108300050') (the WSA address was obtained from the dump report in Figure 181 on page 499). The result (X'108300090') 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 183 shows the sections of the dump.

```
LIST 01_08300050. ASID(X'00CC') LENGTH(X'0100') AREA
_8300050. C360E6E2 C1F8F440 40404040 40404040 |C_KSA64
_8300060. 94B1B995 006A495 83B10000 00000000 |main.funcA......
_8300070. 00000001 08300900 00000000 20900000 |
_8300080. 00000000 00000000 00000000 00000000 |20E71F88 ...........X.8
_8300090. LENGTH(X'10')----All bytes contain X'00'
_83000A0. 00000001 08300200 00000000 0830048B |
_83000C0. 00000001 083004F5 00000000 08300532 |
_83000E0. 00000001 0830056F 00000001 083005AC |
_83000F0. 00000001 083005E9 00000001 08300626 |
_8300100. 00000001 083005F8 00000001 08300663 |
_8300110 LENGTH(X'40')----All bytes contain X'00'
```

Figure 183. IPCS storage display of the writeable static area (AMODE 64)

### 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()` or `vfork()`, or one of the exec family of functions, the SYMDUMP DD allocation information is not inherited. Even though the SYMDUMP allocation is not inherited, a SYMDUMP 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 395.

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 184 on page 502 is a sample filled-in panel that shows the characteristics defined for the URCOMP:JRUSL.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 396 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) run-time 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) run-time option, the storage report displays HeapPools statistics. For a sample storage...
HEAPPOOLS64 storage statistics
The HEAPPOOLS64 run-time 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 run-time 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 run-time option

Element Size
the size of the cell plus any additional storage needed for control information or to maintain alignment

Cells PerExtent
the cell pool count specified by the HEAPPOOLS64 run-time 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

HEAPPOOLS storage statistics
The HEAPPOOLS run-time 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 run-time 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 run-time 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
HeapPool 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 run-time
  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 run-time
  option
Cells Per Extent — the number of cells per extent. This number is calculated using the following formula, with a minimum of four cells:

\[
\text{Initial Heap Size} \times \left(\frac{\text{Extent Percent}}{100}\right) / \text{Element Size}
\]

Note: Having a small number of cells per extent is not suggested because the pool can allocate many extents, which causes the HeapPool 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:

\[
\left(\text{Maximum Cells Used} \times \text{Element Size} \times 100\right) / \text{Initial Heap Size}
\]
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.

For more information about stack and heap storage, see z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode.

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 185 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.

For more information on the __uheapreport() function, see z/OS XL C/C++ Run-Time Library Reference. For tuning tips, see z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode.

---

Storage Report for Enclave Wed Jan 26 20:29:08 2010
Language Environment V01 R13.00

HeapPools Statistics:
Pool 1 size: 32
   Successful Get Heap requests: - 15
Pool 2 size: 128
   Successful Get Heap requests: - 15
Pool 3 size: 512
   Successful Get Heap requests: - 15
   Pool 4 size: 2048
   Successful Get Heap requests: - 15
Pool 5 size: 8192
   Successful Get Heap requests: - 15
Pool 6 size: 16384
   Successful Get Heap requests: - 15
Requests greater than the largest cell size: 0

HeapPools Summary:

<table>
<thead>
<tr>
<th>Cell Size</th>
<th>Extents Allocated</th>
<th>Cells Used</th>
<th>Cells In Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>15</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>128</td>
<td>15</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>512</td>
<td>15</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2048</td>
<td>15</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>8192</td>
<td>15</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>16384</td>
<td>15</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

Suggested Cell Sizes:
(32,128,512,2048,8192,16384,16384,0)

End of Storage Report

---

Figure 185. Storage report generated by __uheapreport() (AMODE 64)
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++ run-time component of Language Environment: 5000 (except for 5500, which are from the DSECT utility), 6000, and 7000.

   - **IGZ**  The prefix for messages from the COBOL run-time component of Language Environment.

   - **FOR**  The prefix for messages from the Fortran run-time component of Language Environment.

   - **IBM**  The prefix for messages from the PL/I run-time component of Language Environment.

   - **CEE**  The prefix for messages from the common run-time component of Language Environment.

4. For any messages received, check for recommendations in the “Programmer Response” sections of the messages in this 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 35 or Chapter 12, "Using Language Environment AMODE 64 debugging facilities," on page 377.
   • 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-specific 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.

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.
   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 the PPA2 at X'04' from the start of the PPA1.
   b. Find the address of the compilation unit's primary entry point at X'10' in the PPA2.
   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 186 on page 510](#)).
   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 187 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.

<table>
<thead>
<tr>
<th></th>
<th>Length of name</th>
<th>Untruncated entry/label name</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>A (PPA2)</td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td>A (Block Debugging Information (BDI)) or zero</td>
<td></td>
</tr>
</tbody>
</table>

Figure 186. C PPA1

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 following URL:


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 67 describes how to produce
documentation required for submission with the APAR.

Table 67. 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 35 (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 run-time options, in machine-readable form</td>
<td>Application programmer</td>
</tr>
<tr>
<td>8</td>
<td>System output associated with the MSGFILE run-time 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

Publications for this product are offered in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when using PDF files, you may view the information through the z/OS Internet Library website or the z/OS Information Center. If you continue to experience problems, send an email to mhvrcfs@us.ibm.com or write to:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to [z/OS TSO/E Primer](http://www.ibm.com/systems/z/os/zos/bserv/) and [z/OS ISPF User’s Guide Vol I](http://www.ibm.com/systems/z/os/zos/bserv/) for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer or Library Server versions of z/OS books in the Internet library at:


One exception is command syntax that is published in railroad track format, which is accessible using screen readers with the Information Center, as described in "Dotted decimal syntax diagrams."

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always
present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 *FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* *FILE indicates that syntax element *FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only
one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

* means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:
1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

+ means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.
Notices

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

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" at [http://www.ibm.com/legal/copytrade.shtml](http://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.
Bibliography

This section lists the books in the Language Environment library and other publications that may be helpful when using Language Environment.

## Language Products Publications

### z/OS Language Environment
- z/OS Language Environment Concepts Guide, SA22-7567
- z/OS Language Environment Programming Guide, SA22-7561
- z/OS Language Environment Programming Reference, SA22-7562
- z/OS Language Environment Customization, SA22-7564
- z/OS Language Environment Debugging Guide, GA22-7560
- z/OS Language Environment Run-Time Application Migration Guide, GA22-7565
- z/OS Language Environment Writing Interlanguage Communication Applications, SA22-7563
- z/OS Language Environment Run-Time Messages, SA22-7566
- z/OS Language Environment Vendor Interfaces, SA22-7568
- z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode, SA22-7569

### z/OS XL C/C++
- z/OS XL C/C++ Language Reference, SC09-4815
- z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer, GC09-4913
- z/OS XL C/C++ Programming Guide, SC09-4765
- z/OS XL C/C++ User’s Guide, SC09-4767
- z/OS XL C/C++ Run-Time Library Reference, SA22-7821
- z/OS XL C/C++ Messages, GC09-4819
- Standard C++ Library Reference, SC09-4949

### z/OS Metal C Runtime Library
- z/OS Metal C Programming Guide and Reference, SA23-2225

### Enterprise COBOL for z/OS, V4R2
- Enterprise COBOL for z/OS, V4R2, Licensed Program Specifications, GI11-7871
- Enterprise COBOL for z/OS, V4R2, Customization, SC23-8526
- Enterprise COBOL for z/OS, V4R2, Language Reference, SC23-8528
- Enterprise COBOL for z/OS, V4R2, Programming Guide, SC23-8529

### Enterprise COBOL for z/OS, V3R4
- Enterprise COBOL for z/OS, V3R4, Licensed Program Specifications, GC27-1411
- Enterprise COBOL for z/OS, V3R4, Customization, GC27-1410
- Enterprise COBOL for z/OS, V3R4, Language Reference, SC27-1408
- Enterprise COBOL for z/OS, V3R4, Programming Guide, SC27-1412

### COBOL for OS/390 & VM
- COBOL for OS/390 & VM Licensed Program Specifications, GC26-9044
- COBOL for OS/390 & VM Customization under OS/390, GC26-9045
Related Publications

CICS
- CICS Transaction Server for z/OS Installation Guide, GC34-6224
- CICS Operations and Utilities Guide, SC34-6229
- CICS Problem Determination Guide, SC34-6239
- CICS Resource Definition Guide, SC34-6228
- CICS Application Programming Reference, SC34-6231
- CICS Application Programming Guide, SC34-6232
- CICS System Definition Guide, SC34-6226

DB2
- Database 2 Application Programming and SQL Guide, SC26-4377

DFSMS/MVS
- z/OS DFSMS RM Guide and Reference, SC26-7396

IPCS
- z/OS MVS IPCS User’s Guide, SA22-7596
- z/OS MVS IPCS Commands, SA22-7594
- z/OS MVS IPCS Customization, SA22-7595
DFSORT
- z/OS DFSORT Application Programming Guide SC26-7523

IMS/ESA
- IMS Version 8: Application Programming: Database Manager SC27-1286
- IMS Version 8: Application Programming: Transaction Manager SC27-1289
- IMS Version 8: Application Programming: EXEC DLI Commands for CICS and IMS Version 8 SC27-1288

z/OS
- z/OS Introduction and Release Guide GA22-7502
- z/OS Program Directory GI10-0670
- z/OS Planning for Installation GA22-7504
- z/OS Information Roadmap SA22-7500
- z/OS Hot Topics Newsletter GA22-7501
- z/OS Licensed Program Specifications GA22-7503

z/OS ISPF
- z/OS ISPF Dialog Tag Language Guide and Reference SC34-4824
- z/OS ISPF Planning and Customizing GC34-4814
- z/OS ISPF Dialog Developer’s Guide and Reference SC34-4821

z/OS UNIX System Services
- z/OS UNIX System Services User’s Guide SA22-7801
- z/OS UNIX System Services Command Reference SA22-7802
- z/OS UNIX System Services Programming: Assembler Callable Services Reference SA22-7803
- z/OS UNIX System Services Planning GA22-7800

z/OS TSO/E
- z/OS TSO/E Customization SA22-7783
- z/OS TSO/E Programming Services SA22-7789
- z/OS TSO/E System Programming Command Reference SA22-7793
Index

Special characters

- abend 176, 464
- alloc 176, 464
- amrc 176, 464
- cabend() function 369, 375, 394
- code 176, 464
- error 176, 464
- feedback 176, 464
- last_op 176, 464
- le_cib_get() function 369
- le_message_get_and_and_write() function 362, 371
- le_message_get() function 371
- le_msg_write() function 362, 371
- msg 177, 466
- reset_exception_handler() function 369
- set_exception_handler() function 369, 370
- BPXK_MDUMP 83, 395
- EDC_ADD_ERRNO2 182, 472

Numerics

64-bit applications 361, 373, 377, 463

A

abend codes
  >= 4000 30, 374
  < 4000 30, 374
  4093 373
  passing to operating system 24
  system, example of 32, 375
  user-specified, example of 32, 375
  user, example of 32, 375, 376
  using 32, 375

abends
  internal, table of output 355
  Language Environment 32, 351, 375
  requested by assembler user exit 24
  system 32, 376
  under CICS 351
  user 32, 376

ABPERC run-time option
  function 8
  generating a system dump and 81
  modifying condition handling
  behavior and 21

ABTERMENC run-time option 8, 24
  using 24

accessibility 515

AGGREGATE compiler option 4, 7

AMODE 64 applications
  classifying errors 373
  debugging C/C++ 463
  preparing for debugging 361
  using debugging 377
  anywhere heap
  statistics 18

APAR (Authorized Program Analysis Report) 511
  documentation 512
  application program interfaces (API) 369, 371
  application programs
  debugging
  handling a storage dump written to a BFS file 231, 501
  handling a storage dump written to an HFS file 231, 501
  argument
  in dump 63
  arguments, registers, and variables for active routines 61, 393
  assembler language
  user exit 23, 24
  for CICS 355
  generating a system dump with 81, 394
  modifying condition handling behavior and 24
  using 23, 24
  atexit information in dump 205
  automatic variables
  locating in dump 300, 332

B

base locator
  for working storage 252
  in dump 252
  below heap
  statistics 18
  BLOCKS option of CEE3DMP callable service 37

C

C library function
  trace table entries for 485
  C return codes to CICS 352
  C-CAA (C-specific common anchor area)
  See C/C++, C-specific common anchor area (C-CAA)
  C/C++
  __amrc
  example of structure 176, 464
  information in dump 206
  __msg 177, 466
  atexit
  information in dump 205
  C-specific common anchor area (C-CAA)
  cdump() function 193, 477
  compiler options 3
  debugging examples 221, 228, 487
  dump information in 205

C/C++ (continued)
  dump (continued)
  parameter in 189
  signal information in 205
  structure variables, locating in 190
  system, structures in 190
  file
  control block information 206
  status and attributes in dump 206
  functions
  calling dump, example 192, 476
  cdump() 42, 192, 379, 476
  csnap() 42, 192, 194, 379, 476, 477
  ctrace() 42, 192, 194, 379, 476, 477
  fetch() 175, 463
  fopen() 177, 466
  perror() 175, 182, 463, 471
  printf() 176, 464
  to produce dump output 42, 379
  memory file control block 206
  stdio.h 176, 464
  timestamp 192
  CAA (common anchor area) 65, 393,
  398, 400, 413, 445
  call chain 63

CALL statement
  CDUMP/CPDUMP 273
  DUMP/PDUMP 272
  SDUMP 273
  callable services 20

CEE3ABD—terminate enclave with an abend
  See CEE3ABD—terminate enclave with an abend

CEE3DMP—generate dump
  See CEE3DMP—generate dump

CEE3GRO—returns location offset
  See CEE3GRO—returns location offset

CEE3SRP—set resume point
  See CEE3SRP—set resume point

CEEDCOD—decompose a condition token
  See CEEDCOD—decompose a condition token

CEEHDLR—register user condition handler
  See CEEHDLR—register user condition handler

CEEMGET—get a message
  See CEEMGET—get a message

CEEMOUT—dispatch a message
  See CEEMOUT—dispatch a message

CEEMRCE—move resume cursor to a designated label
  See CEEMRCE—move resume cursor to designated label
Callable Services (continued)
CEEMRCR—move resume cursor relative to handle cursor
See CEEMRCR—move resume cursor relative to handle cursor
CEEMSG—get, format, and dispatch a message
See CEEMSG—get, format, and dispatch a message
CEESGL—signal a condition
See CEESGL—signal a condition

case 1 condition token 27, 371
case 2 condition token 27, 371
cdmp() function 192, 374, 476
class test 244
classifying errors table 29, 373
CLLE (COBOL load list entry) 251
COBECOM control block 256
COBOL
base locator for working storage 252
compiler options 5
dumping examples 256, 269
dump
external data in 252
file information in 252
linkage section in 252
local variables in 249
routine information in 249
run unit storage in 254
stack frames for active routines in 249
working storage in 252
errors 243
listings 247
memory file control block 205
program class storage 252
return codes to CICS 352
routine
calling Language Environment dump service 247
COBVEC control block 256
command
syntax diagrams xv
COMMAREA (Communication Area) 351
compiler options
C 3
COBOL 5
Fortran 5
LP64 361, 463
PL/I 6
condition
information for active routines 61, 392
in dump 61, 392
POSIX 27
unhandled 27, 371
condition handling behavior, modifying 20
user-written condition handler 20, 23
condition information block (CIB) 63, 74
condition manager 27, 371
CONDITION option of CEE3DMP
callable service 38, 63
condition token 26, 27, 370
case 1 27, 371
case 2 27, 371
equality of 27, 371
conditions, nested 28, 372
control block
for active routines 61, 393
csnap() function 194, 477
ctrace() function 194, 374, 477
D
data
map listing 247
values 63
DCB (data control block) 251
DCT (destination control table) 349
DEBUG run-time option 8
dumping
C, examples 221, 228, 487
COBOL, examples 256, 269
for CICS 349
Fortran, examples 278, 283
PL/I, examples 304, 309, 336, 344
tool 35, 377
DEPTHCNDLMT run-time option
function 8, 362
modifying condition handling behavior and 21
wait/loop error and 30
diagnosis checklist 507
disability 515
display
errnojr_value 33
DISPLAY statement 25, 243
displays
errnojr_value 33
divide-by-zero error 221, 262, 280, 309,
314, 344, 348, 487
DSA (dynamic save area) 63
See stack, frame
dummy DSA 63
dump
an area of storage 273
date in 57, 390
dynamically allocated storage in 62
storage written to an HFS file 231, 501
symbolic 273
DUMP suboption of TERMTHDACT
run-time option 39, 377
DUMP/PDUMP routine 272
format specifications 272
output 272
usage considerations 272
E
ECB (enclave control block) 62, 393
EDB (enclave data block) 62, 393
EDC prefix 29, 31, 373, 375
edcmttext shell command 33
eib (exec interface block) 351
enclave
identifier in dump 58, 391
member list 62, 63, 393, 394
storage 62
termination
behavior, establishing 24
entry information 57, 390
ENTRY option of CEE3DMP callable service 38
entry point
name of active routines in dump 391
ERRCOUNT run-time option
function 8, 9
modifying condition handling behavior and 22
wait/loop error and 30
errno 205
ermnr_value
displaying 33
error
determining source of 507
message while Language Environment was handling another error 28, 372
unanticipated 29, 373
ESD compiler option 7
examples
application abends from 356
C routines 221, 228, 487
calling a nonexistent subroutine 259, 307, 340
COBOL routines 256, 269
divide-by-zero error 221, 262, 280, 309, 314, 344, 348, 487
Fortran routines 278, 283
output under CICS 349
PL/I routines 304, 309, 336, 344
SUBSCRIPTRANGE error 256, 304, 336
exception handling behavior, modifying 369
user-written exception handler 369
EXEC CICS DUMP statements 351
external data
for COBOL programs in dump 252
fetch
fetch information in dump 205
fetch() function 175, 463
fetchable module 175, 463
file
for COBOL, in dump 252
status key 244
file control block (FCB) 206
FILES option of CEE3DMP callable service 37
FLAG compiler option 4
floating point registers in dump 58
fopen() function 177, 466
FOR prefix 29, 31
Fortran
compiler options 5
debugging examples 278, 283
dump services 271
events, determining the source of 269
Fortran (continued)
lisings 271
G
general purpose registers 58
GMAREA 251
GONUMBER compiler option 4
H
HANDLE ABEND EXEC CICS command 349
header files, C
cctest.h 192, 476
errno.h 221, 487
stdio.h 176, 464
stdlib.h 221, 487
heap storage
created by CEECRHP callable service 20
in LEDATA Output 112, 415
reports 116, 423
storage in dump 62
user 17
HEAP64 run-time option 363
HEAPCHK run-time option
function 9, 117, 362, 425
HEAPPOOLS
storage statistics 233, 234, 504, 505
user-created, __uheapreport() 505
user-created, _uheapreport 234
trace report 87, 123, 398, 431
HEAPPOOLS run-time option 363
HEAPPOOLS64
storage statistics 503
HEAPPOOLS64 run-time option 363
I
I/O
conventions 175, 463
IBM prefix 29, 31
IGZ prefix 29, 31
INFOMSGFILTER run-time option
function 9, 362
INITIALIZE statement 244
instance specific information (ISI) 371
instruction length counter (ILC) in dump 61, 392
Inter-procedural Analysis (IPA) 361
interactive problem control system (IPCS)
analyzing a storage dump 231, 501
cbf command 451
VERBEXIT 397, 401, 435
INTERRUPT compiler option
function 7
INTERRUPT run-time option 8, 9
interruption code in dump 61, 392
IOHEAP64 run-time option 363
ITBLK in dump 256
K
keyboard 515
L
LAA (library anchor area) 453
language constructs 243
Language Environment
return codes to CICS 352
symbolic feedback code 26, 370
Language Environment dump
C information in 204
CEEDUMP 8, 35, 361
COBOL information in 251
default options 36
eexample traceback in 62, 394
Fortran information in 276
multiple enclaves and options
BLOCKS 37
CONDITION 38, 63
ENCLAVE 37
ENTRY 38
FILES 37
FNAME 38
NOBLOCKS 37
NOCONDITION 38
NOENTRY 38
NOFILES 37
NOSTORAGE 37
NOTRACEBACK 37
NOVARIABLES 37
PAGESIZE(n) 38
STACKFRAME 38
STORAGE 37
THREAD 37
TRACEBACK 37, 63
VARIABLES 37, 63
output
for C routines 198
for COBOL program 247
for Fortran routines 276
for PL/I routines 292, 326
information for multiple enclaves 42, 379
PL/I information in 295, 329
section descriptions 56, 390
TERMTHDACT suboptions 40, 378
title 57, 390
traceback with condition information
C routine 198, 480
COBOL program 249
Fortran routine 269, 277
Language Environment routine 56, 390
PL/I routine 292, 326
using C functions 42, 379
using CDUMP/CPDUMP subroutine 271
using CEE3DMP callable service 35, 56, 390
using DUMP/PDUMP subroutine 271
using PLIDUMP subroutine 42, 292, 326
using SDUMP subroutine 271
using TERMTHDACT run-time option 38, 377
LCA (library communication area) 453