
This is a major revision of GA22-7560-09.

This edition applies to Language Environment in z/OS Version 1 Release 11 (5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. A form for readers’ comments may be provided at the back of this document, or you may address your comments to the following address:

International Business Machines Corporation
MHVRCFS, Mail Station P181
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+845+432-9405
FAX (Other Countries):
   Your International Access Code +1+845+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
Internet e-mail: mhvrdfs@us.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:
• Title and order number of this document
• Page number or topic related to your comment

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

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
Part 2. Debugging language-specific routines ................................. 165

Chapter 4. Debugging C/C++ routines ........................................... 167
Debugging C/C++ programs ....................................................... 167
Using the __amrc and __amrc2 structures to debug input/output ........................................ 167
Using file I/O tracing to debug C/C++ file I/O problems .......................................................... 174
Displaying an error message with the perror() function ......................................................... 174
Using __errno2() to diagnose application problems ................................................................. 174
Diagnosing DLL problems ......................................................................................................... 176
Using C/C++ listings ................................................................................................................. 176
Finding variables ...................................................................................................................... 177
Generating a Language Environment dump of a C/C++ routine .............................................. 185
cdump() ....................................................................................................................................... 185
csnap() ....................................................................................................................................... 186
ctrace() ....................................................................................................................................... 186
Sample C routine that calls cdump() ....................................................................................... 186
Sample C++ routine that generates a Language Environment dump ........................................ 188
Sample Language Environment dump with C/C++-specific information ................................ 190
Finding C/C++ information in a Language Environment dump ............................................... 196
Sample Language Environment dump with XPLINK-specific information ............................ 201
Finding XPLINK information in a Language Environment dump ............................................ 206
C/C++ contents of the Language Environment trace tables ..................................................... 207
Debugging examples of C/C++ routines ..................................................................................... 214
Divide-by-zero error .................................................................................................................. 214
Calling a nonexistent non-XPLINK function .......................................................................... 218
Calling a nonexistent XPLINK function .................................................................................. 221
Handling dumps written to the z/OS UNIX file system ............................................................ 224
Multithreading consideration .................................................................................................... 225
Understanding C/C++ heap information in storage reports .................................................... 226
Language Environment storage report with HEAPPOOLS statistics ....................................... 226
C function __uheapreport() storage report ............................................................................. 227
MEMCHECK VHM memory leak analysis tool .......................................................................... 229
Invoking MEMCHECK VHM ...................................................................................................... 229
MEMCHECK VHM environment variables .............................................................................. 230
MEMCHECK VHM report sample scenario .............................................................................. 231
MEMCHECK VHM report examples .......................................................................................... 232

Chapter 5. Debugging COBOL programs .................................................................................. 237
Determining the source of error ............................................................................................... 237
Tracing program logic .............................................................................................................. 237
Finding input/output errors ..................................................................................................... 238
Handling input/output errors .................................................................................................... 238
Validating data (class test) ........................................................................................................ 238
Assessing switch problems ...................................................................................................... 238
Generating information about procedures .............................................................................. 239
Using COBOL listings ............................................................................................................... 241
Generating a Language Environment dump of a COBOL program ....................................... 241
COBOL program that calls another COBOL program .............................................................. 241
COBOL program that calls the Language Environment CEE3DMP callable service ............... 242
Sample Language Environment dump with COBOL-specific information ................................ 243
Finding COBOL information in a dump .................................................................................. 245
Debugging example COBOL programs .................................................................................... 250
Subscript range error ................................................................................................................ 250
Calling a nonexistent subroutine .............................................................................................. 253
Divide-by-zero error .................................................................................................................. 256

Chapter 6. Debugging Fortran routines .................................................................................... 263
Determining the source of errors in Fortran routines ............................................................... 263
Identifying run-time errors ........................................................................................................ 263
Using Fortran compiler listings ........................................ 265
Generating a Language Environment dump of a Fortran routine .... 265
DUMP/PDUMP subroutines ............................................. 266
CDUMP/CPDUMP subroutines ........................................... 267
SDUMP subroutine ..................................................... 268
Finding Fortran information in a Language Environment dump .... 271
Understanding the Language Environment traceback table ......... 272
Examples of debugging Fortran routines ............................... 273
Calling a nonexistent routine ......................................... 273
Divide-by-zero error .................................................... 275

Chapter 7. Debugging PL/I for MVS & VM routines .................... 279
Determining the source of errors in PL/I for MVS & VM routines . 279
  Logic errors in the source routine ................................ 279
  Invalid use of PL/I for MVS & VM ................................. 279
  Unforeseen errors .................................................. 280
  Invalid input data .................................................. 280
  Compiler or run-time routine malfunction ......................... 280
  System malfunction ................................................ 280
  Unidentified routine malfunction ................................ 280
  Storage overlay problems ........................................... 281
Using PL/I for MVS & VM compiler listings ........................... 282
  Generating PL/I for MVS & VM listings and maps ................. 282
  Finding information in PL/I for MVS & VM listings ............... 283
Generating a Language Environment dump of a PL/I for MVS & VM routine ..................................................... 289
  PLIDUMP syntax and options ...................................... 289
  PLIDUMP usage notes .............................................. 290
Finding PL/I for MVS & VM information in a dump .................... 291
  Traceback .......................................................... 291
  Control blocks for active routines ................................ 294
  Control blocks associated with the thread ......................... 298
PL/I for MVS & VM contents of the Language Environment trace table ..................................................... 300
Debugging example of PL/I for MVS & VM routines ..................... 301
  Subscript range error ............................................. 301
  Calling a nonexistent subroutine .................................. 305
  Divide-by-zero error .............................................. 307

Chapter 8. Debugging Enterprise PL/I routines ......................... 315
Determining the source of errors in Enterprise PL/I routines ....... 315
  Logic errors in the source routine ................................ 315
  Invalid use of Enterprise PL/I ..................................... 315
  Unforeseen errors .................................................. 316
  Invalid input data .................................................. 316
  Compiler or run-time routine malfunction ......................... 316
  System malfunction ................................................ 316
  Unidentified routine malfunction ................................ 316
  Storage overlay problems ........................................... 317
Using Enterprise PL/I compiler listings ................................ 318
  Generating Enterprise PL/I listings and maps ..................... 318
  Finding information in Enterprise PL/I listings ................. 319
Generating a Language Environment dump of an Enterprise PL/I routine ..................................................... 324
  PLIDUMP syntax and options ...................................... 325
  PLIDUMP usage notes .............................................. 326
Finding Enterprise PL/I information in a dump ....................... 327
  Traceback .......................................................... 327
  Control blocks for active routines ................................ 329
# 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 ................................................................................. 44
6. The C DLL CELDLL ......................................................................................... 47
7. Example dump using CEE3DMP ........................................................................ 48
8. Upward-growing (non-XPLINK) stack frame format ......................................... 64
9. Downward-growing (XPLINK) stack frame format ............................................. 65
10. Common anchor area ....................................................................................... 66
11. Condition information block ......................................................................... 75
12. Machine state information block ................................................................ 78
13. Language Environment dump of multiple enclaves ......................................... 80
14. Example of formatted output from LEDATA Verbexit .................................... 90
15. Example of formatted PTBL output from LEDATA Verbexit .............................. 106
16. Example formatted detailed heap segment report from LEDATA Verbexit .... 108
17. Example formatted detailed heappool report from LEDATA Verbexit ............... 114
18. Example formatted detailed HEAPPOLS trace report from LEDATA Verbexit .... 119
19. Example formatted C/C++ output from LEDATA Verbexit .............................. 124
20. Example formatted COBOL output from LEDATA Verbexit ............................. 148
21. The CAA formatted by the CBFORMAT IPCS command .................................... 151
22. Format of the trace table entry ...................................................................... 154
23. Trace table in dump output ............................................................................ 162
24. _amrc structure ............................................................................................ 168
25. _amrc2 structure ......................................................................................... 169
26. Example of a routine using perror() ............................................................. 174
27. Example of a routine using __errno2() ......................................................... 175
28. Sample output of a routine using __errno2() ............................................... 175
29. Example of a routine using _EDC_ADD_ERRNO2 ....................................... 175
30. Sample output of a routine using _EDC_ADD_ERRNO2 .................................. 175
31. Example of a routine using __err2ad() in combination with __errno2() .... 176
32. Sample output of routine using __err2ad() in combination with __errno2() ...... 176
33. Writable static map produced by prelinker ..................................................... 179
34. Location of RENT static variable in storage .................................................. 180
35. Writable static map produced by prelinker ..................................................... 180
36. Location of NORENT static variable in storage ............................................. 181
37. Example code for parameter variable ............................................................ 182
38. Example code for parameter variable ............................................................ 182
39. Partial storage offset listing ......................................................................... 183
40. Example code for structure variable .............................................................. 183
41. Example of aggregate map .......................................................................... 184
42. Writable static map produced by prelinker ..................................................... 184
43. Example C routine using cdump() to generate a dump ................................. 187
44. Fetched module for C routine ........................................................................ 188
45. Example C++ routine with protection exception generating a dump ............ 189
46. Template file STACK.C ................................................................................. 189
47. Header file STACK.H ..................................................................................... 190
48. Example dump from sample C routine ........................................................... 191
49. Memory file control block ............................................................................ 199
50. Registers on entry to CEE3DMP .................................................................... 200
51. Parameters, registers, and variables for active routines ............................... 200
52. Condition information for active routines ..................................................... 201
53. Sample XPLINK-compiled program (tranmain) which calls a NOXPLINK-compiled program 202
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>How to Use z/OS Language Environment Publications</td>
<td>xvii</td>
</tr>
<tr>
<td>2.</td>
<td>Syntax examples</td>
<td>xx</td>
</tr>
<tr>
<td>3.</td>
<td>Common error symptoms, possible causes, and programmer responses</td>
<td>30</td>
</tr>
<tr>
<td>4.</td>
<td>CEE3DMP options</td>
<td>36</td>
</tr>
<tr>
<td>5.</td>
<td>TERMTHDACT suboptions, level of information, and destinations</td>
<td>39</td>
</tr>
<tr>
<td>6.</td>
<td>Condition handling of 0Cx abends</td>
<td>42</td>
</tr>
<tr>
<td>7.</td>
<td>List of CAA fields</td>
<td>67</td>
</tr>
<tr>
<td>8.</td>
<td>Language Environment Control blocks that can be individually formatted</td>
<td>151</td>
</tr>
<tr>
<td>9.</td>
<td>LE=1 entry records</td>
<td>155</td>
</tr>
<tr>
<td>10.</td>
<td>LE=2 entry records</td>
<td>156</td>
</tr>
<tr>
<td>11.</td>
<td>Format of the mutex/CV/latch records</td>
<td>160</td>
</tr>
<tr>
<td>12.</td>
<td>LE=8 entry records</td>
<td>161</td>
</tr>
<tr>
<td>13.</td>
<td>LE=20 entry records</td>
<td>161</td>
</tr>
<tr>
<td>14.</td>
<td>__last_op values and diagnosis information</td>
<td>170</td>
</tr>
<tr>
<td>15.</td>
<td>Finding the WSA base address</td>
<td>177</td>
</tr>
<tr>
<td>16.</td>
<td>Compiler-generated COBOL listings and their contents</td>
<td>241</td>
</tr>
<tr>
<td>17.</td>
<td>Compiler-generated Fortran listings and their contents</td>
<td>265</td>
</tr>
<tr>
<td>18.</td>
<td>Compiler-generated PL/I for MVS &amp; VM listings and their contents</td>
<td>282</td>
</tr>
<tr>
<td>19.</td>
<td>Typical comments in a PL/I for MVS &amp; VM static storage listing</td>
<td>285</td>
</tr>
<tr>
<td>20.</td>
<td>Comments in a PL/I for MVS &amp; VM object code listing</td>
<td>287</td>
</tr>
<tr>
<td>21.</td>
<td>PL/I for MVS &amp; VM mnemonics</td>
<td>288</td>
</tr>
<tr>
<td>22.</td>
<td>Compiler-generated PL/I listings and their contents</td>
<td>318</td>
</tr>
<tr>
<td>23.</td>
<td>Finding data when Language Environment returns a nonzero return code</td>
<td>353</td>
</tr>
<tr>
<td>24.</td>
<td>Finding data when Language Environment abends internally</td>
<td>353</td>
</tr>
<tr>
<td>25.</td>
<td>Finding data when Language Environment abends from an EXEC CICS command</td>
<td>354</td>
</tr>
<tr>
<td>26.</td>
<td>Common error symptoms, possible causes, and programmer responses</td>
<td>372</td>
</tr>
<tr>
<td>27.</td>
<td>TERMTHDACT suboptions, level of information, and destinations</td>
<td>375</td>
</tr>
<tr>
<td>28.</td>
<td>Language Environment control blocks which can be individually formatted</td>
<td>463</td>
</tr>
<tr>
<td>29.</td>
<td>Preinitalized Environments for Authorized Programs control blocks which can be individually formatted</td>
<td>464</td>
</tr>
<tr>
<td>30.</td>
<td>LE=1 entry records</td>
<td>467</td>
</tr>
<tr>
<td>31.</td>
<td>LE=2 entry records</td>
<td>467</td>
</tr>
<tr>
<td>32.</td>
<td>Format of the mutex/CV/latch records</td>
<td>471</td>
</tr>
<tr>
<td>33.</td>
<td>LE=8 entry records</td>
<td>472</td>
</tr>
<tr>
<td>34.</td>
<td>__last_op values and diagnosis information</td>
<td>479</td>
</tr>
<tr>
<td>35.</td>
<td>Problem resolution documentation requirements</td>
<td>526</td>
</tr>
</tbody>
</table>
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.

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 "Bibliography" on page 535.

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</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>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>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.

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>
<tr>
<td>Repeatable item.</td>
</tr>
<tr>
<td>Fragment.</td>
</tr>
</tbody>
</table>
This debugging guide

*z/OS Language Environment Debugging Guide* provides assistance with detecting and locating 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 for application programmers 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 above
- Program storage concepts

Where to find more information

Please see *z/OS Information Roadmap* for an overview of the documentation associated with z/OS.

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):
Summary of Changes

Summary of Changes
for GA22-7560-10
z/OS Version 1 Release 11

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

The following summarizes the changes to that information.

New Information

- "Understanding the heap pool LEDATA output" on page 113 has been added in Chapter 3, "Using Language Environment debugging facilities," on page 35.
- "Understanding the heap pool LEDATA output" on page 429 has been added in Chapter 12, "Using Language Environment AMODE 64 debugging facilities," on page 375.
- "Requesting a UNIX System Services syscall trace for debugging" on page 163 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 473 has been added in Chapter 12, "Using Language Environment AMODE 64 debugging facilities," on page 375.
- "Using file I/O tracing to debug C/C++ file I/O problems" on page 174 has been added in Chapter 4, "Debugging C/C++ routines," on page 167.
- "Using file I/O tracing to debug C/C++ file I/O problems" on page 482 has been added in Chapter 13, "Debugging AMODE 64 C/C++ routines," on page 475.

Changed Information

- Figure 1 on page 10 has been updated in "Determining run-time options in effect" on page 9.
- Figure 12 on page 78 has been updated in "Using the machine state information block" on page 77.
- Figure 14 on page 90 has been updated in "Understanding the Language Environment IPCS Verbexit LEDATA output" on page 89.
- Figure 138 on page 361 has been updated in "Determining run-time options in effect" on page 360.
- Condition information for active routine has been updated in "Sections of the Language Environment dump" on page 56.
- CAA field, CEECAA_CICS_EXT_REG, has been updated in Table 7 on page 67.
- In "Language Environment IPCS Verbexit – LEDATA" on page 85, the Condition Management Control Blocks section in Chapter 3, "Using Language Environment debugging facilities," on page 35 has been updated.
- In "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 375 has been updated.

You may notice changes in the style and structure of some content in this document—for example, headings that use uppercase for the first letter of initial
words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

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

Summary of Changes for GA22-7560-09
z/OS Version 1 Release 10

This document contains information previously presented in z/OS Language Environment Debugging Guide, GA22-7560-08, which supported z/OS Version 1 Release 9.

The following summarizes the changes to that information.

**New Information**

- "HEAPPOOLS storage statistics" on page 517 has been added to "Language Environment storage report with heap pools statistics" on page 516.

**Changed Information**

- A usage note has been added to "Language Environment IPCS Verbexit – LEDATA" on page 85 and "Language Environment IPCS Verbexit – LEDATA" on page 397.
- In "Understanding the C/C++-specific LEDATA output" on page 123, Figure 19 on page 124 has been updated with changed and new sections.
- In "Understanding the C/C++-specific LEDATA output" on page 442, Figure 149 on page 443 has been updated with changed and new sections.
- A new parameter has been added to "Language Environment IPCS Verbexit – LEDATA" on page 85.
- In "Determining run-time options in effect" on page 360, the figure of 64–bit options report has been updated.
- In "Controlling storage allocation" on page 11, Figure 3 on page 14 has been replaced.
- In "HEAPPOOLS storage statistics" on page 226, HEAPPOOLS statistics has been updated.
- In "HEAPPOOLS64 storage statistics" on page 516, HEAPPOOLS64 statistics and HEAPPOOLS64 summary has been updated.
- In "Understanding the HEAPPOOLS trace LEDATA output" on page 118, Figure 18 on page 119 has been updated.
- In "Understanding the heap pools trace LEDATA output" on page 435, Figure 148 on page 436 has been updated.
- "Storage statistics for AMODE 64 applications" on page 365 has been moved from Appendix C in z/OS Language Environment Programming Reference to "Controlling storage allocation" on page 361.

This document contains terminology, maintenance, and editorial changes, including changes to improve consistency and retrievability.
Summary of Changes
for GA22-7560-08
z/OS Version 1 Release 9

This document contains information previously presented in z/OS Language Environment Debugging Guide, GA22-7560-07, which supported z/OS Version 1 Release 8.

The following summarizes the changes to that information.

New Information
• A new chapter, Chapter 8, “Debugging Enterprise PL/I routines,” on page 315, has been added to provide information about debugging Enterprise PL/I routines.
• Language Environment® improves DLL error diagnostics so that you can more easily debug DLL application problems. For more information, see “Diagnosing DLL problems” on page 176 and “Using the DLL failure control block” on page 78.
• A new section, “Using edcmtext to obtain information about errno2 values” on page 33, has been added to provide the edcmtext utility (similar to bpxmtext), which allows faster error resolution when an errno2 is encountered in Language Environment.

Changed Information
• In Chapter 3, “Using Language Environment debugging facilities,” on page 35, the following sections have been updated:
  – “Debug tools” on page 35
  – “Understanding the Language Environment dump” on page 43
  – “The Common Anchor Area” on page 65
  – “Formatting and analyzing system dumps” on page 84
• In Chapter 4, “Debugging C/C++ routines,” on page 167, the following sections have been updated:
  – “Using __errno2() to diagnose application problems” on page 174
  – “Sample Language Environment dump with C/C++-specific information” on page 190
  – “Sample Language Environment dump with XPLINK-specific information” on page 201
  – “Debugging examples of C/C++ routines” on page 214
  – “Understanding C/C++ heap information in storage reports” on page 226, example storage reports and the description of the HeapPools storage statistics section of the storage report are updated.
• In Chapter 5, “Debugging COBOL programs,” on page 237, figures have been updated in the following sections:
  – “Sample Language Environment dump with COBOL-specific information” on page 243
  – “Finding COBOL information in a dump” on page 245
  – “Debugging example COBOL programs” on page 250
• In Chapter 7, “Debugging PL/I for MVS & VM routines,” on page 279, the following sections have been updated:
  – “Finding PL/I for MVS & VM information in a dump” on page 291
  – “Debugging example of PL/I for MVS & VM routines” on page 301
• In Chapter 9, “Debugging under CICS,” on page 347, the Locating the Language Environment traceback section has been updated; enhancements are made in the CICS CLER transaction for displaying and modifying Language Environment run-time options. For more information, see “Displaying and modifying run-time options with the CLER transaction” on page 354.

• In Chapter 10, “Preparing your AMODE 64 application for debugging,” on page 359, Figure 138 on page 361 and Figure 139 on page 362 have been updated.

• In Chapter 12, “Using Language Environment AMODE 64 debugging facilities,” on page 375, the following sections have been updated:
  – “Understanding the Language Environment dump” on page 377
  – “Language Environment IPCS Verbexit – LEDATA” on page 397
  – “Understanding the Language Environment IPCS Verbexit LEDATA output” on page 401
  – “Understanding the HEAP LEDATA output” on page 418
  – “Formatting individual control blocks” on page 462

• In Chapter 13, “Debugging AMODE 64 C/C++ routines,” on page 475, the following sections have been updated:
  – “Using __errno2() to diagnose application problems” on page 483
  – “Calling a nonexistent function” on page 507
  – “Understanding C/C++ heap information in storage reports” on page 515

This document contains terminology, maintenance, and editorial changes, including changes to improve consistency and retrievability.
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, which is equivalent to TEST(LINE,BLOCK,PATHT,SYM,HOOK). For XL C++, the option TEST is equivalent to TEST(HOOK). Following is a list of TEST suboptions that you can use to simplify run-time debugging.

| ALL | Sets all of the TEST suboptions. |
| BLOCK | Generates symbol information for nested blocks. |
| HOOK | Generates all possible hooks. For details on this suboption, see [z/OS XL C/C++ User's Guide](#). |
| LINE | Generates line number hooks and allows a debugging tool to generate a symbolic dump. |
| PATH | 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. |
| SYM | 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:

```
%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 these C/C++ compiler options to make run-time debugging easier:

**AGGREGATE (C)** Specifies that a layout for `struct` and `union` type variables appear in the listing.

**ATTRIBUTE (C++)** For XL C++ compile, cross reference listing with attribute information. If XREF is specified, the listing also contains reference, definition and modification information.

**CHECKOUT (C)** Provides informational messages indicating possible programming errors.

**EVENTS** Produces an events file that contains error information and source file statistics.

**EXPMAC** Macro expansions with the original source.

**FLAG** Specifies the minimum severity level that is tolerated.

**GONUMBER** 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.

**INFO (C++)** Indication of possible programming errors.

**INLINE** Inline Summary and Detailed Call Structure Reports. (Specify with the REPORT sub-option).

**INLRPT** Generates a report on status of functions that were inlined. The OPTIMIZE option must also be specified.

**LIST** Listing of the pseudo-assembly listing produced by the compiler.

**OFFSET** Displays the offset addresses relative to the entry point of each function.

**PHASEID** Causes each compiler module (phase) to issue an informational message which identifies the compiler phase module name, product identifier, and build level.

**PPONLY** 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 “#include” and “#define” directives.

**SERVICE** Places a string in the object module, which is displayed in the traceback if the application fails abnormally.

**SHOWINC** All included text in the listing.

**SOURCE** Includes source input statements and diagnostic messages in the listing.

**TERMINAL** Directs all error messages from the compiler to the terminal. If not specified, this is the default.

**TEST** Generates information for debugging interface. This also generates symbol tables needed for symbolic variables in the dump.

**XPLINK (BACKCHAIN)** Generates a prolog that saves redundant information in the calling function’s stack frame.

**XPLINK (STOREARGS)** Generates code to store arguments that are normally passed in registers, into the argument area.

**XREF** 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.

For a detailed explanation of these options, see the [z/OS XL C/C++ User’s Guide](#).

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 following COBOL compiler options to prepare your program for run-time debugging:

- **LIST**: 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.
- **MAP**: Produces lists of items in data division including a data division map, global tables, literal pools, a nested program structure map, and attributes.
- **OFFSET**: Produces a condensed PROCEDURE DIVISION listing containing line numbers, statement references, and location of the first instruction generated for each statement.
- **OUTDD**: Specifies the destination of DISPLAY statement messages.
- **SOURCE**: Produces a listing of your source program with any statements embedded by PROCESS, COPY, or BASIS statements.
- **TEST**: 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.
- **VBREF**: Produces a cross-reference of all verb types used in the source program and a summary of how many times each verb is used.
- **XREF**: Creates a sorted cross-reference listing.

For more detail on these options and functions, see the following documents:
- *Enterprise COBOL for z/OS Programming Guide*, SC23-8529
- *COBOL for OS/390 & VM Programming Guide*

**Fortran compiler options**

You can use these Fortran compiler options to prepare your program for run-time debugging:

- **FIPS**: Specifies whether standard language flagging is to be performed. This is valuable if you want to write a program conforming to Fortran 77.
- **FLAG**: 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.
- **GOSTMT**: Specifies that statement numbers are included in the run-time messages and in the Language Environment dump.
ICA
Specifies whether 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.

LIST
Specifies whether 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.

MAP
Specifies whether a table of source program variable names, named constants, and statement labels and their displacements is to be produced.

OPTIMIZE
Specifies the optimizing level to be used during compilation. If you are debugging your program, it is advisable to use NOOPTIMIZE.

SDUMP
Specifies whether dump information is to be generated.

SOURCE
Specifies whether a source listing is to be produced.

SRCFLG
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.

SXM
Formats SREF or MAP listing output to a 72-character width.

SYM
Invokes the production of SYM cards in the object text file. SYM cards contain location information for variables within a Fortran program.

TERMINAL
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.

TEST
Specifies whether to override any optimization level above OPTIMIZE(0). This option adds run-time overhead.

TRMFLG
Specifies whether to display the initial line of source statements in error and their associated error messages at the terminal.

XREF
Creates a cross-reference listing.

VECTOR
Specifies whether to invoke the vectorization process. A vectorization report provides detailed information about the vectorization process.

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.

### 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 these compiler options to prepare PL/I routines for debugging:

**AGGREGATE**
Specifies that a layout for arrays and major structures appears in the listing.

**ESD**
Includes the external symbol dictionary in the listing.

**GONUMBER/GOSTMT**
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.

INTERRUPT Specifies that users can establish an ATTENTION ON-unit that gains control when an attention interrupt occurs.

LIST 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.

LMESSAGE 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.

MAP 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.

OFFSET 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.

SOURCE Specifies that the compiler includes a listing of the source routine in the listing.

STORAGE Includes a table of the main storage requirements for the object module in the listing.

TERMINAL Specifies what parts of the compiler listing produced during compilation are directed to the terminal.

TEST 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.

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

For more detail on PL/I compiler options, see PL/I for MVS & VM Programming Guide.

Enterprise PL/I for z/OS compiler options

The following Enterprise PL/I for z/OS compiler options can be specified when preparing your Enterprise PL/I for z/OS routines for debugging:

AGGREGATE Specifies that a layout for arrays and major structures appears in the listing.

GONUMBER / GOSTMT 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.

INTERRUPT Specifies that users can establish an ATTENTION ON-unit that gains control when an attention interrupt occurs.

LIST 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.

OFFSET Displays the offset addresses relative to the entry point of each function.

SOURCE Specifies that the compiler includes a listing of the source routine in the listing.

STORAGE Includes a table of the main storage requirements for the object module in the listing.
TEST
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.

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

For further details on the Enterprise PL/I for z/OS compiler options, see Enterprise PL/I for z/OS Programming Guide.

Using Language Environment run-time options
There are several run-time options that 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.

The following Language Environment run-time options affect debugging:

- **ABPERC**
  Specifies that the indicated abend code bypasses the condition handler.

- **ABTERMENC**
  Specifies enclave termination behavior for an enclave ending with an unhandled condition of severity 2 or greater.

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

- **CHECK**
  Determines whether run-time checking is performed.

- **NODEBUG**
  Controls the COBOL USE FOR DEBUGGING declarative.

- **DEPTHCONDLMT**
  Specifies the limit for the depth of nested synchronous conditions in user-written condition handlers. (Asynchronous signals do not affect DEPTHCONDLMT.)

- **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.

- **ERRCOUNT**
  Specifies the number of synchronous conditions of severity 2 or greater tolerated. (Asynchronous signals do not affect ERRCOUNT.)

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

- **INFOMSGFILTER**
  Filters user specified informational messages from the MSGFILE.
  **Note:** Affects only those messages generated by Language Environment and any routine that calls CEEMSG. Other routines that write to the message file, such as CEEMOUT, do not have a filtering option.

- **INTERRUPT**
  Causes Language Environment to recognize attention interrupts.

- **MSGFILE**
  Specifies the ddname of the Language Environment message file.

- **MSGQ**
  Specifies the number of instance specific information (ISI) blocks that are allocated on a per-thread basis for use by an application. Located within the Language Environment condition token is an ISI token. The ISI contains information used by the condition manager to identify and react to a specific occurrence of a condition.
PROFILE
Controls the use of an optional profiler tool, which collects performance data for the running application. When this option is in effect, the profiler is loaded and the debugger cannot be loaded. If the TEST option is in effect when PROFILE is specified, the profiler tool will not be loaded.

RPTOPTS
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 11.

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 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.

USRHDLR
Specifies the behavior of two user-written condition handlers. The first handler specified will be registered at stack frame 0. The second handler specified will be registered before any other user-written condition handlers, once the handler is enabled by a condition.

XUFLOW
Specifies whether or not an exponent underflow causes a routine interrupt.

For a more detailed discussion of these run-time options, see z/OS Language Environment Programming Reference.

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 on page 10 shows a sample options report.
### Options Report for Enclave main 12/07/08 6:25:56 PM
Language Environment V01 R11.00

<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>CBLPPTS(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CBLPSHPPOP(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>NODEBUG</td>
</tr>
<tr>
<td>Installation default</td>
<td>DEPTHCONDLMT(10)</td>
</tr>
<tr>
<td>Installation default</td>
<td>DYNHEAP(*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>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>PARMLIB(CEEPRMPV)</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>HEAPPOOLS(OFF,16,384,128,0,1,0,10,0,1024,0,2048,0,1024,0,10)</td>
</tr>
<tr>
<td>Installation default</td>
<td>INFOMSGFILTER(OFF,,,)</td>
</tr>
<tr>
<td>Installation default</td>
<td>INTERRUPT(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>INQPCOPN</td>
</tr>
<tr>
<td>Installation default</td>
<td>LIBSTACK(4096,4096,FREE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>MSGFILE(SYSOUT,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>NONONIPTSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>Installation default</td>
<td>OCSTATUS</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOPC</td>
</tr>
<tr>
<td>Installation default</td>
<td>PLITASKCOUNT(20)</td>
</tr>
<tr>
<td>Installation default</td>
<td>POSIX(OFF)</td>
</tr>
<tr>
<td>Installation default</td>
<td>PROFILE(OFF,&quot;&quot;)</td>
</tr>
<tr>
<td>Installation default</td>
<td>PRUNIT(6)</td>
</tr>
<tr>
<td>Installation default</td>
<td>PUNIT(5)</td>
</tr>
<tr>
<td>Installation default</td>
<td>RDRUNIT(5)</td>
</tr>
<tr>
<td>Installation default</td>
<td>RECPAD(OFF)</td>
</tr>
<tr>
<td>Invocation command</td>
<td>RPTOPTS(ON)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>RPTSTG(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NORTEREUS</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOUSIMVRD</td>
</tr>
<tr>
<td>Installation default</td>
<td>STACK(131072,131072,ANYWHERE,KEEP,524288,131072)</td>
</tr>
<tr>
<td>Installation default</td>
<td>STORAGE(NONE,NONE,NONE,0)</td>
</tr>
<tr>
<td>Installation default</td>
<td>TERMTHDACT(TRACE,.,96)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOTEST(*,,&quot;PRIM&quot;,&quot;INSPPREF&quot;)</td>
</tr>
<tr>
<td>Installation default</td>
<td>THREADHEAP(4096,4096,ANYWHERE,KEEP)</td>
</tr>
<tr>
<td>Installation default</td>
<td>THREADSTACK(OFF,4096,4096,ANYWHERE,KEEP,131072,131072)</td>
</tr>
<tr>
<td>Installation default</td>
<td>TRACE(OFF,4096,DUMP,LE=0)</td>
</tr>
<tr>
<td>Installation default</td>
<td>TRAP(ON,SPIE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>UPSI(90000000)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOUSRDLR(,)</td>
</tr>
<tr>
<td>Installation default</td>
<td>VCTRSAVE(OFF)</td>
</tr>
<tr>
<td>Installation default</td>
<td>XPLINK(OFF)</td>
</tr>
<tr>
<td>Installation default</td>
<td>XUFLOW(AUTO)</td>
</tr>
</tbody>
</table>

---

**Figure 1. Options report example produced by run-time option RPTOPTS(ON)**
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 354.

Controlling storage allocation

The following run-time options control storage allocation:

- STACK
- THREADSTACK
- LIBSTACK
- THREADHEAP
- HEAP
- ANYHEAP
- BELOWHEAP
- STORAGE
- HEAPPOOLS

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

To generate a report of the storage a routine (or more specifically, an enclave) used during its run, specify the RPTSTG(ON) 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 on page 12 and Figure 3 on page 14 show sample storage reports. The sections that follow these reports describe the contents of the reports.
### Storage Report for Enclave main 12/18/08 5:17:20 PM
Language Environment V01 R11.00

#### STACK statistics:
- **Initial size:** 4096
- **Increment size:** 4096
- **Maximum used by all concurrent threads:** 7488
- **Largest used by any thread:** 7488
- **Number of segments allocated:** 2
- **Number of segments freed:** 0

#### THREADSTACK statistics:
- **Initial size:** 4096
- **Increment size:** 4096
- **Maximum used by all concurrent threads:** 3352
- **Largest used by any thread:** 3352
- **Number of segments allocated:** 6
- **Number of segments freed:** 0

#### LIBSTACK statistics:
- **Initial size:** 4096
- **Increment size:** 4096
- **Maximum used by all concurrent threads:** 0
- **Largest used by any thread:** 0
- **Number of segments allocated:** 0
- **Number of segments freed:** 0

#### THREADHEAP statistics:
- **Initial size:** 4096
- **Increment size:** 4096
- **Maximum used by all concurrent threads:** 0
- **Largest used by any thread:** 0
- **Successful Get Heap requests:** 0
- **Successful Free Heap requests:** 0
- **Number of segments allocated:** 0
- **Number of segments freed:** 0

#### HEAP statistics:
- **Initial size:** 49152
- **Increment size:** 16384
- **Total heap storage used (sugg. initial size):** 29112
- **Successful Get Heap requests:** 251
- **Successful Free Heap requests:** 218
- **Number of segments allocated:** 1
- **Number of segments freed:** 0

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

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

---

*Figure 2. Storage report produced by run-time option RPTSTG(ON) (Part 1 of 2)*
BELOWHEAP statistics:
Initial size: 8192
Increment size: 8192
Total heap storage used (sugg. initial size): 0
Successful Get Heap requests: 0
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0
Additional Heap statistics:
Successful Create Heap requests: 1
Successful Discard Heap requests: 1
Total heap storage used: 4912
Successful Get Heap requests: 3
Successful Free Heap requests: 3
Number of segments allocated: 2
Number of segments freed: 2
Largest number of threads concurrently active: 2

End of Storage Report

Figure 2. Storage report produced by run-time option RPTSTG(ON) (Part 2 of 2)
### STACK statistics:
- Initial size: 131072
- Increment size: 131072
- Maximum used by all concurrent threads: 5416
- Largest used by any thread: 5416
- Number of segments allocated: 1
- Number of segments freed: 0

### THREADSTACK statistics:
- Initial size: 4096
- Increment size: 4096
- Maximum used by all concurrent threads: 45536
- Largest used by any thread: 6552
- Number of segments allocated: 60
- Number of segments freed: 0

### XPLINK STACK statistics:
- Initial size: 524288
- Increment size: 131072
- Largest used by any thread: 20400
- Number of segments allocated: 1
- Number of segments freed: 0

### XPLINK THREADSTACK statistics:
- Initial size: 131072
- Increment size: 131072
- Largest used by any thread: 22160
- Number of segments allocated: 30
- Number of segments freed: 0

### LIBSTACK statistics:
- Initial size: 4096
- Increment size: 4096
- Maximum used by all concurrent threads: 0
- Largest used by any thread: 0
- Number of segments allocated: 0
- Number of segments freed: 0

### THREADHEAP statistics:
- Initial size: 4096
- Increment size: 4096
- Maximum used by all concurrent threads: 0
- Largest used by any thread: 0
- Successful Get Heap requests: 0
- Successful Free Heap requests: 0
- Number of segments allocated: 0
- Number of segments freed: 0

### HEAP statistics:
- Initial size: 32768
- Increment size: 32768
- Total heap storage used (sugg. initial size): 286576
- Successful Get Heap requests: 71
- Successful Free Heap requests: 1
- Number of segments allocated: 10
- Number of segments freed: 0

---

**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
- 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
- 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

---

<table>
<thead>
<tr>
<th>Pool 4</th>
<th>Size: 256 Get Requests: 38</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful Get Heap requests: 137- 144</td>
<td>2</td>
</tr>
<tr>
<td>Successful Get Heap requests: 145- 152</td>
<td>2</td>
</tr>
<tr>
<td>Successful Get Heap requests: 153- 160</td>
<td>2</td>
</tr>
<tr>
<td>Successful Get Heap requests: 161- 168</td>
<td>1</td>
</tr>
<tr>
<td>Successful Get Heap requests: 169- 176</td>
<td>4</td>
</tr>
<tr>
<td>Successful Get Heap requests: 177- 184</td>
<td>4</td>
</tr>
<tr>
<td>Successful Get Heap requests: 185- 192</td>
<td>2</td>
</tr>
<tr>
<td>Successful Get Heap requests: 193- 200</td>
<td>2</td>
</tr>
<tr>
<td>Successful Get Heap requests: 201- 208</td>
<td>3</td>
</tr>
<tr>
<td>Successful Get Heap requests: 209- 216</td>
<td>2</td>
</tr>
<tr>
<td>Successful Get Heap requests: 217- 224</td>
<td>3</td>
</tr>
<tr>
<td>Successful Get Heap requests: 225- 232</td>
<td>3</td>
</tr>
<tr>
<td>Successful Get Heap requests: 233- 240</td>
<td>1</td>
</tr>
<tr>
<td>Successful Get Heap requests: 241- 248</td>
<td>3</td>
</tr>
<tr>
<td>Successful Get Heap requests: 249- 256</td>
<td>4</td>
</tr>
</tbody>
</table>

| Pool 5.1 | Size: 1024 Get Requests: 230 |
| Pool 5.2 | Size: 1024 Get Requests: 3 |
| Pool 5.3 | Size: 1024 Get Requests: 5 |
| Successful Get Heap requests: 257- 264 | 225 |
| Successful Get Heap requests: 273- 280 | 2 |
| Successful Get Heap requests: 281- 288 | 10 |
| Successful Get Heap requests: 841- 848 | 1 |

| Pool 6 | Size: 2048 Get Requests: 2 |
| Successful Get Heap requests: 1113- 1120 | 1 |
| Successful Get Heap requests: 1137- 1144 | 1 |

Requests greater than the largest cell size: 2

HEAPPOLLS Summary:

<table>
<thead>
<tr>
<th>Specified Element</th>
<th>Extent</th>
<th>Cells Per Extent</th>
<th>Maximum Cells Allocated</th>
<th>Cells In Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Size Size</td>
<td>Percent</td>
<td>Extents</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>204</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>40</td>
<td>81</td>
<td>3</td>
<td>226</td>
</tr>
<tr>
<td>128</td>
<td>10</td>
<td>24</td>
<td>4</td>
<td>88</td>
</tr>
<tr>
<td>256</td>
<td>10</td>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>4</td>
<td>57</td>
<td>228</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2048</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Suggested Percentages for current Cell Sizes:

HEAPP(ON,8,132,28128,37,256,1,(1024,3),90,2048,13,0)

Suggested Cell Sizes:

HEAPP(ON,32,56,88,120,128,168,208,248,288,848,1144,2080)

Largest number of threads concurrently active: 11

End of Storage Report

---

**Figure 3. Storage report produced by RPTSTG(ON) with XPLINK (Part 3 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 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.

STACK, THREADSTACK, LIBSTACK, and IPT STACK statistics for the upward-growing stack

- Initial size — the 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.)
- Increment size — the size of each incremental segment acquired, 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 segments (or increments) allocated — the number of incremental segments allocated by all threads.
- Number of segments freed — the 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.

XPLINK statistics — XPLINK STACK and XPLINK THREADSTACK statistics for the downward-growing stack

These sections of the storage report only apply if XPLINK is in effect.

- Initial size — the actual size of the initial segment assigned to each thread.
- Increment size — the size of each incremental segment acquired, 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 segments allocated — the number of incremental segments allocated by all threads.
- Number of segments freed — the 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.

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.

**THREADHEAP statistics**

- Initial size—the default initial allocation, as specified by the corresponding run-time option. Please note that for HEAP24, the initial size is the value of the initpsz24 of the HEAP option.
- Increment size—the minimum incremental allocation, as specified by the corresponding run-time option. Please note that for HEAP24, the increment size is the value of the incrsz24 of the HEAP option.
- Maximum used by all concurrent threads—the maximum total amount used by all concurrent threads at any one time.
- Largest used by any thread—the largest amount used by any single thread.
- 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.

These statistics, in all cases, specify totals for the whole enclave. For THREADHEAP, they indicate the total across all threads in the enclave.
HEAP, HEAP24, ANYHEAP, and BELOWHEAP statistics

- Initial size—the default initial allocation, as specified by the corresponding run-time option. Please note that for HEAP24, the initial size is the value of the initsz24 of the HEAP option.
- Increment size—the minimum incremental allocation, as specified by the corresponding run-time option. Please note that for HEAP24, the increment size is the value of the incrsz24 of the HEAP 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 freed individually, but rather were freed implicitly in the course of enclave termination.

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.

These statistics, in all cases, specify totals for the whole enclave. For THREADHEAP, they indicate the total across all threads in the enclave.

Additional heap statistics

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

- Successful Create Heap requests—the number of successful Create Heap requests.
- Successful Discard Heap requests—the 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.

- 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.

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 226.

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

You can use these callable services to modify condition handling:

- **CEE3ABD** Terminates an enclave using an abend.
- **CEE3AB2** Terminate enclave with an abend and reason code.
- **CEEMRCE** Moves the resume cursor to an explicit location where resumption is to occur after a condition has been handled.
- **CEEMRCC** Moves the resume cursor relative to the current position of the handle cursor.
- **CEE3CIB** Returns a pointer to a condition information block (CIB) associated with a given condition token. The CIB contains detailed information about the condition.
- **CEE3GRO** Returns the offset of the location within the most current Language Environment-conforming routine where a condition occurred.
- **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.

Fortran programs cannot directly call Language Environment callable services. For more information about callable services, see [z/OS Language Environment Programming Reference](http://www.ibm.com). For more information about how to invoke callable services from Fortran, see [Language Environment for MVS & VM Fortran Run-Time Migration Guide](http://www.ibm.com).

Language Environment run-time options

These Language Environment run-time options can affect your routine’s condition handling behavior:
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 hhh is a hexadecimal system abend code.

User abends are specified as Udddd, where dddd is a decimal user abend code. Any other 4-character EBCDIC string, such as NONE, that is not of the form Shhh can also be specified as a user-specified abend code. You can specify only one abend code with this option. This option assumes the use of TRAP(ON). ABPERC is not supported in CICS.

Language Environment ignores ABPERC(0Cx). No abend is percolated and Language Environment condition handling semantics are in effect.

CHECK

Specifies that checking errors within an application are detected. The Language Environment-conforming languages can define error checking differently.

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').

ERRCOUNT

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.

INTERRUPT

Causes attentions recognized by the host operating system to be passed to and recognized by Language Environment after the environment has been initialized.

TERMTHDACT

Sets the level of information that is produced when a condition of severity 2 or greater remains unhandled within the enclave. There are five possible parameter settings for different levels of information:

- 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 condition handler. This causes the Language Environment condition 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), it is still possible to invoke the condition handler through the CEESGL callable service and pass conditions to registered user-written condition handlers.

Specify TRAP(OFF) when you do not want Language Environment to issue an ESTAE or an ESPIE.

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. For further information, see the TRAP run-time option in z/OS Language Environment Programming Reference.

USRHDLR Specifies the behavior of two user-written condition handlers. The first handler specified will be registered at stack frame 0. The second handler specified will be registered before any other user-written condition handlers, once the handler is enabled by a condition.

When you specify USRHDLR(lastname, supername), lastname gets control at stack frame 0. Supername will get control first, before any user-written condition handlers but after supername has gone through the enablement phase, when a condition occurs.

XUFLOW Specifies whether an exponent underflow causes a routine interrupt.

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 CEEAUE_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 runtime, 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 30.)

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 27).

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. The following table lists these callable services and their functions.

<table>
<thead>
<tr>
<th>Callable Service</th>
<th>Function</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.
For example, in the condition token: X'0003032D 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

Figure 4. Language Environment condition token

For example, in the condition token: X'0003032D 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 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 \textit{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 2/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 3 lists common error symptoms, possible causes, and programmer responses.

Table 3. 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 &quot;Interpreting run-time messages&quot; below.</td>
</tr>
<tr>
<td>User abend code &lt; 4000</td>
<td>a) A non-Language Environment abend occurred</td>
<td>See the Language Environment abend codes in z/OS Language Environment Run-Time Messages. Check for a subsystem-generated abend or a user-specified abend.</td>
</tr>
<tr>
<td></td>
<td>b) The assembler user exit requested an abend for an unhandled condition of severity ≥2</td>
<td></td>
</tr>
<tr>
<td>User abend code ≥ 4000</td>
<td>a) Language Environment detected an error and could not proceed</td>
<td>For any abends you receive, read the appropriate explanation listed in the abend codes section of z/OS Language Environment Run-Time Messages.</td>
</tr>
<tr>
<td></td>
<td>b) An unhandled software-raised condition occurred and ABTERMENC(ABEND) was in effect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) The assembler user exit requested an abend for an unhandled condition of severity 4</td>
<td></td>
</tr>
<tr>
<td>System abend with TRAP(OFF)</td>
<td>Cause depends on type of malfunction</td>
<td>Respond appropriately. See the messages and codes book of the operating system.</td>
</tr>
<tr>
<td>System abend with TRAP(ON)</td>
<td>System-detected error</td>
<td>See the messages and codes book 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 DEPTHCONDLMNT 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 ddbname, 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 z/OS Language Environment Programming Guide</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 ddd 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 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

z/OS UNIX® environment:

edcmtext errno2_value

TSO/E environment:

EDCMTEXT errno2_value

Description

edcmtext 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.

errno2_value is specified as 8 hexadecimal characters.

The user can specify one of the following in place of a errno2 value to view a help dialog: -h, help, ?.

The user can specify the -U option to display the output in uppercase.

Usage notes

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'CFFFFFFF'.

The utility bpxmtext displays the description and action text for reason codes returned from the kernel, in addition to errno2_values returned from the C/C++ run-time library. The user should use bpxmtext when the source of the errno2_values is unknown. For further information about bpxmtext, see z/OS UNIX System Services Command Reference.

Message returns

If the user specifies a -h, help or ? in place of the errno2_value, the following message is displayed:

Usage: edcmtext errno2_value

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

erno2_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 the environment that edcmtext is being run in is not TSO/E or z/OS UNIX 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:
JrEdclopsEinval01: 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: edclopst.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 http://www-306.ibm.com/software/awdtools/debugtool/tools/wddsz/.

You can also use dbx to debug Language Environment applications, including C/C++ programs. [z/OS UNIX System Services Command Reference](http://www.ibm.com/support/docview.wss?rs=180&uid=swg27013767) has information on dbx subcommands, while [z/OS UNIX System Services Programming Tools](http://www.ibm.com/support/docview.wss?rs=180&uid=swg27013767) 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.

Below is a list of methods to invoke the Language Environment dump service:

- 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.

**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.

Following is a list of CEE3DMP options and related information:

<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>Dump Options</td>
<td>Abbreviation</td>
<td>Action Taken</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------</td>
<td>------------------------------------------------------------------------------</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>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 either 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 (see below). 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 via 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>
Table 4. 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 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>Generate a run-time options report in the dump output. This will be the default if an unhandled condition occurs, and a CEE3DUMP is generated due to the setting of the TERMTHDACT run-time option setting.</td>
</tr>
<tr>
<td>NOGENOPTS</td>
<td>NOGENO</td>
<td>Do 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>

**Note:** On CICS, only ENCLAVE(CURRENT) and ENCLAVE(1) settings are supported.

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)

For additional information about the CEE3DMP callable service and dump options, see [z/OS Language Environment Programming Reference](https://www.ibm.com/support/knowledgecenter/en/SSLTBK_22r0124/scale/coshand ledit.html). For an example of a Language Environment dump, see "Understanding the Language Environment dump" on page 43.
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](https://www.ibm.com/support/docview.wss?uid=swg27047455).

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 5. TERMTHDACT suboptions, level of information, and destinations**

<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of Information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIET</td>
<td>No information</td>
<td>No destination.</td>
</tr>
<tr>
<td>MSG</td>
<td>Message</td>
<td>Terminal or ddname specified in MSGFILE 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>
</tbody>
</table>
Table 5. TERMTHDACT suboptions, level of information, and destinations (continued)

<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of Information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>UADUMP</td>
<td>Message, Language Environment dump, and SYSDUMP, 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 CEEEDUMP 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:
- 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
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.

- **COBOL Considerations**
  - The following TERMTHDACT suboptions for COBOL are recommended, UAONLY, UATRACE, and UADUMP. A system dump will always be generated when one of these suboptions is specified.

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

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

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

- **CICS Considerations**
  - TERMTHDACT output is written either 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 6 on page 42 shows the behavior of CESE|CICSDDS when they are used with the other suboptions of TERMTHDACT.
  - Since 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.
<table>
<thead>
<tr>
<th>Options</th>
<th>TERMTHDACT(X,CESE,)</th>
<th>TERMTHDACT(X,CICSDDS,)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIET</td>
<td>• No output.</td>
<td>• No output.</td>
</tr>
<tr>
<td>MSG</td>
<td>• Message written to CESE queue or MSGFILE.</td>
<td>• Message written to CESE queue or MSGFILE.</td>
</tr>
</tbody>
</table>
| TRACE   | • The traceback is written to the CESE queue, followed by U4038 abend with nodump option. | • Language Environment will write traceback, variables, COBOL working storage, C writeable static. The member handlers will be invoked to provide the desired output to the new transaction server queue (which CICS will read and write to CICS transaction dump later).  
  • U4039 abend to force CICS transaction dump followed by U4038 abend with nodump option.  
  • Message to CESE or MSGFILE. |
| DUMP    | • CEDUMP to CESE queue followed by U4038 abend with nodump option. | • CEDUMP to new transaction server queue which CICS will read and write to CICS transaction dump later.  
  • U4039 abend to force CICS transaction dump followed by U4038 abend with nodump option.  
  • Message to CESE or MSGFILE. |
| UATRACE | • U4039 abend with traceback to CESE queue followed by U4038 abend with nodump option. | • Language Environment will write traceback, variables, COBOL working storage, C writeable statics. The member handlers will be invoked to provide the desired output to the new transaction server queue (which CICS will read and write to CICS transaction dump later).  
  • U4039 abend to force CICS transaction dump followed by U4038 abend with nodump option.  
  • Message to CESE or MSGFILE. |
| UADUMP  | • U4039 abend with CEDUMP to CESE queue followed by U4038 abend with nodump option. | • CEDUMP 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 CEDUMP 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 CEDUMP information is generated.  
  • Same as CESE. |
**Note:** Program checks and other abends will cause CICS to produce a CICS transaction dump.

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

### 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 185.

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 265.

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 289.

### 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 48](#) 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 44](#). CELSAMP uses the DLL CELDLL shown in [Figure 6 on page 47](#).
#pragma options(SERVICE("1.1.d"),NOOPT,TEST(SYM))
#pragma runopts(TERMTHDACT(UADUMP),POSIX(ON),DYNDUMP(DYNAMIC,))
#pragma runopts(TRACE(ON,1M,NODUMP,LE=1),HEAPCHK(ON,1,0,10,10))
#pragma runopts(RPTSTG(GON),HEAPPOOLS(ON))
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <dll.h>
#include <signal.h>
#include <leawi.h>
#include <ceeedcct.h>

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

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");
}

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[1], 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...
");
pgmptr.nesting = NULL;
token = 2;
CEEHDLR (&pgmptr, &token, &fc);
if (_FBCHECK ( fc , CEE000 ) != 0 ) {
    printf("CEEHDLR failed with message number %d\n", fc.tok_msgno);
    exit(108);
}

printf("Write to some files...
");
fp1 = fopen("myfile.data", "w");
if (!fp1) {
    perror("Could not open myfile.data for write");
    exit(109);
}

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

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

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

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

Figure 5. The C program CELSAMP (Part 3 of 3)
/* 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 <ceedcct.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 (OC9) */
    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

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.
Figure 7. Example dump using CEE3DMP (Part 1 of 9)
Figure 7. Example dump using CEE3DMP (Part 2 of 9)
Figure 7. Example dump using CEE3DMP (Part 3 of 9)
Figure 7. Example dump using CEE3DMP (Part 4 of 9)
Figure 7. Example dump using CEE3DMP (Part 5 of 9)
## Enclave Control Blocks:

<table>
<thead>
<tr>
<th>EDB</th>
<th>20914648</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 20914648</td>
<td></td>
</tr>
<tr>
<td>C3C5C5C5C4C24040 C7000100 20915870 20914050 00000000 00000000 00000000</td>
<td>CEEEDB...</td>
</tr>
<tr>
<td>+000020 20914648</td>
<td></td>
</tr>
<tr>
<td>20914648 20914648 20914648 20914648 20914648 20914648 20914648 00000000</td>
<td>...j...h...y...q...j...</td>
</tr>
<tr>
<td>+000040 20914648</td>
<td></td>
</tr>
<tr>
<td>20914648 20914648 20914648 20914648 20914648 20914648 20914648 00000000</td>
<td>...r.y...r.y...</td>
</tr>
<tr>
<td>+000060 20914648</td>
<td></td>
</tr>
<tr>
<td>0000F460 21070000 2193E5E8 00000000 20916550 20966600 20915980 20915980</td>
<td>...&amp;...j... &amp;...j...</td>
</tr>
<tr>
<td>+000080 20914648</td>
<td></td>
</tr>
<tr>
<td>20914648 90000000 0000A5F6 00000000 00000002 0006FF00 008FF46B 008FF46B</td>
<td>...6.....70..4Y</td>
</tr>
<tr>
<td>+0000A0 20914648</td>
<td></td>
</tr>
<tr>
<td>00000001 00109C9B 2091464F 00000000 00000000 00000000</td>
<td>...h.j...0...</td>
</tr>
</tbody>
</table>

### MEML:
- 20915870
- 20915870
- 20915890
- 00A99180
- 20915890
- 00000000
- 0020948E
- 20915890
- 00000000
- 0020948E
- 20915890
- 00000000
- 0020948E

### Mutex and Condition Variable Blocks
- MCBB=MCMT=MCCT: 210F701B
- MCBB=MCMT=MCCT: 210F701B
- MCBB=MCMT=MCCT: 210F701B
- MCBB=MCMT=MCCT: 210F701B
- MCBB=MCMT=MCCT: 210F701B
- MCBB=MCMT=MCCT: 210F701B
- MCBB=MCMT=MCCT: 210F701B
- MCBB=MCMT=MCCT: 210F701B
- MCBB=MCMT=MCCT: 210F701B
- MCBB=MCMT=MCCT: 210F701B

### Thread Synchronization Enclave Latch Table (EPALT): 210F7544
- 210F7544
- 210F7544
- 210F7544
- 210F7544
- 210F7544
- 210F7544
- 210F7544
- 210F7544
- 210F7544
- 210F7544

### DLL Information:

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### HEAPCHK Option Control Block (HCOP): 20FC0000
- 20FC0000
- 20FC0000
- 20FC0000
- 20FC0000
- 20FC0000
- 20FC0000
- 20FC0000
- 20FC0000
- 20FC0000
- 20FC0000

### HEAPCHK Element Table (HCET) for Heapid 2134FF4C:

<table>
<thead>
<tr>
<th>Header: 21380028</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 21380028</td>
</tr>
<tr>
<td>CBC53C5D 21380028</td>
</tr>
<tr>
<td>00000000 21384FFC</td>
</tr>
<tr>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

### Table: 21380048

| +000000 21380048 |
| +000020 21380048 |
| 00000000 00000000 00000000 21380048 |
| +000020 21380048 |
| 00000000 00000000 00000000 00000000 | HCOP... |

### HEAPCHK Element Table (HCET) for Heapid 21366F4C:

<table>
<thead>
<tr>
<th>Header: 21360028</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 21360028</td>
</tr>
<tr>
<td>CBC53C5D 21360028</td>
</tr>
<tr>
<td>00000000 21366FC4</td>
</tr>
<tr>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

### Table: 21360048

| +000000 21360048 |
| +000020 21360048 |
| 00000000 00000000 00000000 2136C310 |
| +000020 21360048 |
| 00000000 00000000 00000000 00000000 | HCET... |

### HEAPCHK Element Table (HCET) for Heapid 20FC0000:

<table>
<thead>
<tr>
<th>Header: 20FC0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 20FC0000</td>
</tr>
<tr>
<td>CBC53C5D 20FC0000</td>
</tr>
<tr>
<td>00000000 21366F4B</td>
</tr>
<tr>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

### Table: 20FC0000

| +000000 20FC0000 |
| +000020 20FC0000 |
| 00000000 00000000 00000000 00000000 | HCET... |

### WSA address: 20FC0000

### Heap Storage Diagnostics

<table>
<thead>
<tr>
<th>Stg Addr ID</th>
<th>Length</th>
<th>Entry</th>
<th>E Addr</th>
<th>E Offset</th>
<th>Load Mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>2138A020</td>
<td>21384F9C 00000050</td>
<td>20A853F8</td>
<td>+00000000</td>
<td>CEEPLPKA</td>
<td></td>
</tr>
<tr>
<td>CEEVGTS</td>
<td>20A853C8</td>
<td>+0001F84</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEEVGST</td>
<td>20A655E0</td>
<td>+00000027</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>identify_cu</td>
<td>206F778</td>
<td>+0000005A</td>
<td>CEEV003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetDumpsVars</td>
<td>20079000</td>
<td>+00000024</td>
<td>CEEV003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>zdmpd</td>
<td>20E689C0</td>
<td>+0000002E</td>
<td>CEEV003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEEKDMR</td>
<td>209E2838</td>
<td>+00003068</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEEKDMFP</td>
<td>209F8ED0</td>
<td>+0000003C</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEEKDMF</td>
<td>21366EAB</td>
<td>+00000012E</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEEKDMFTN</td>
<td>20A82198</td>
<td>+0000005A</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Chapter 3. Using Language Environment debugging facilities

---

**Figure 7. Example dump using CEE3DMP (Part 6 of 9)**
Figure 7. Example dump using CEE3DMP (Part 7 of 9)
### 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>NOAIXBLD</td>
</tr>
<tr>
<td>Installation default</td>
<td>ALL3I(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>CBLPSHPOP(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CBLTHDA(F0)</td>
</tr>
<tr>
<td>Installation default</td>
<td>CCEEDUMP(60,SYOUT++,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>NODEBUG</td>
</tr>
<tr>
<td>Installation default</td>
<td>DEPTHKONDLMT(10)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>DYNDUMP(POSIX.HEALY.ZOS19,DYNAMIC,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>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,KEEP,8192,4096)</td>
</tr>
<tr>
<td>Programmer default</td>
<td>HEAPCHK(ON,1,0,10,10)</td>
</tr>
<tr>
<td>Programmer default</td>
<td>INFOMSGFILTER(OFF,,,,)</td>
</tr>
<tr>
<td>Programmer default</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>Programmer default</td>
<td>LIBSTACK(4096,4096,FREE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>MSGFILE(SYSOUT,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>NONNIPTSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>Installation default</td>
<td>OCSTATUS</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOSPC</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(OFF,&quot;&quot;)</td>
</tr>
<tr>
<td>Installation default</td>
<td>RDRUNIT(5)</td>
</tr>
<tr>
<td>Installation default</td>
<td>RRUNIT(5)</td>
</tr>
<tr>
<td>Installation default</td>
<td>RPTOFF(OFF)</td>
</tr>
<tr>
<td>Programmer default</td>
<td>RPTSTG(ON)</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOSIMVRD</td>
</tr>
<tr>
<td>Installation default</td>
<td>NOTEST(ALL,&quot;*&quot;,&quot;PROMPT&quot;,&quot;INSPPREF&quot;)</td>
</tr>
<tr>
<td>Installation default</td>
<td>THREADHEAP(4096,4096,ANYWHERE,KEEP)</td>
</tr>
<tr>
<td>Programmer default</td>
<td>THREADSTACK(OFF,4096,4096,ANYWHERE,KEEP,131072,131072)</td>
</tr>
<tr>
<td>Programmer default</td>
<td>TRAP(ON,SPIE)</td>
</tr>
<tr>
<td>Installation default</td>
<td>UPSI(00000000)</td>
</tr>
<tr>
<td>Installation default</td>
<td>XPLINK(OFF)</td>
</tr>
<tr>
<td>Installation default</td>
<td>XUFLOW(AUTO)</td>
</tr>
</tbody>
</table>

---

**Figure 7. Example dump using CEE3DMP (Part 8 of 9)**
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 196.
- For COBOL routines, see “Finding COBOL information in a dump” on page 245.
- For Fortran routines, see “Finding Fortran information in a Language Environment dump” on page 271.
- For PL/I routines, see “Finding PL/I for MVS & VM information in a dump” on page 291.

[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

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 userid associated to the CEEDUMP.

For CEEDUMPs produced under CICS, the following items are displayed:

- **Transaction ID and task number.**

[2] **CEE3845I CEEDUMP Processing started.**

Message CEE3845I 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**

This information identifies the routine name and offset in the calling routine of the call to the dump service.

[4] **Registers on Entry to CEE3DMP**

This section of the dump shows data at the time of the call to the dump service.

- **Program mask**
  
  The program mask contains the bits for the fixed-point overflow mask, decimal overflow mask, exponent underflow mask, and significance mask.

- **General purpose registers (GPRs) 0–15**
  
  On entry to CEE3DMP, the GPRs contain:

<table>
<thead>
<tr>
<th>GPR 0</th>
<th>Working register</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR 1</td>
<td>Pointer to the argument list</td>
</tr>
<tr>
<td>GPR 2–11</td>
<td>Working registers</td>
</tr>
<tr>
<td>GPR 12</td>
<td>Address of CAA</td>
</tr>
<tr>
<td>GPR 13</td>
<td>Pointer to caller’s stack frame</td>
</tr>
<tr>
<td>GPR 14</td>
<td>Address of next instruction to run if the ALL31 run-time option is set to ON</td>
</tr>
<tr>
<td>GPR 15</td>
<td>Entry point of CEE3DMP</td>
</tr>
</tbody>
</table>

- **Floating point registers (FPRs) 0 through 15**
- 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.

These sections show information that is specific to an enclave. When multiple enclaves are dumped, these sections will appear for each enclave.

[5] Enclave Identifier
This statement names the enclave for which information in the dump is provided. If multiple enclaves exist, the dump service generates data and storage information for the most current enclave, followed by previous enclaves in a last-in-first-out (LIFO) order. For more information about dumps for multiple enclaves, see "Multiple enclave dumps" on page 79.

These sections show information that is specific to a thread. When multiple threads are dumped, these sections will appear for each thread.

[6] Information for thread
This section shows the system identifier for the thread. Each thread has a unique identifier.

[7] Traceback
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:
- 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.
- 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.
- Entry point offset
- 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.
- 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.
- 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.

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:

- If your compiled routine is in a partitioned data set, only the member will be output.
- If your compiled routine is in a sequential data set, only the last qualifier will be shown.
- If your compiled routine is in an UNIX filename, only what fits of the filename will be displayed in a line.

• 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' or 'exception'.

The second part contains:

• DSA number
  A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback.

• Stack frame (DSA) address
• Entry point address
• Program unit address
• Program unit offset
  The offset of the last instruction to run in the routine. If the offset is a negative number, zero, or a very large positive number, the routine associated with the offset probably did not allocate a save area or 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.

• Compile Date
  Contains the year, month and day in which the routine was compiled.

• Attributes
  The available compilation attributes of the compile unit including:
  – 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.
  – If the CEEDUMP was created under a POSIX environment, POSIX will be displayed.

The third part of the traceback, 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 is available to Language Environment.

- **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.

[8] **Condition Information for Active Routines**

This section 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
  - 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
  These values are the current values at the time the condition was raised.

[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

[11] Storage for Active Routines

This displays local storage for each active routine. The storage is dumped in hexadecimal, with EBCDIC translations on the right side of the page. There can be other information, depending on the language used. For C/C++ routines, this is the stack frame storage. For COBOL programs, this is language-specific information, WORKING-STORAGE, and LOCAL-STORAGE.

[12] Control Blocks Associated with the Thread

This section 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.

[13] Enclave variables:

This section displays language specific global variables. This section also shows the variable type. Variables are displayed only if the symbol tables are available.

[14] Enclave Control Blocks

This section 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 HEAPCK 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

This section 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.

[17] Run-Time Options Report

This section lists the Language Environment run-time options in effect when the routine was executed.

[18] Process Control Blocks

This section 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:

This section 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.


Message CEE3846I identifies the end of the Language Environment dump processing. Similarly, message CEE3845I identifies the start of the dump processing. Message number CEE3846I can be used to locate the end of the previous CEEDUMP report when scanning backward in a data set that contains several CEEDUMP reports.

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.

The 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 48.

The 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] 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.
The downward-growing (XPLINK) stack frame section

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

Figure 8. Upward-growing (non-XPLINK) stack frame format

The 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](#).

### The 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.

Figure 10 on page 66 shows the format of the Language Environment CAA.

---

**Figure 9. Downward-growing (XPLINK) stack frame format**

<table>
<thead>
<tr>
<th>Low Addresses</th>
<th>Guard Page (4 KB)</th>
<th>Stack Frames for called functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Pointer (R4) +2048</td>
<td>Backchain</td>
<td>Savearea (48 bytes)</td>
</tr>
<tr>
<td></td>
<td>Environment Entry Point Return Address</td>
<td>R8 R9 R10 R11 R12 R13 R14 R15</td>
</tr>
<tr>
<td>+2096</td>
<td>Reserved (8 bytes)</td>
<td></td>
</tr>
<tr>
<td>+2104</td>
<td>Debug Area (4 bytes)</td>
<td></td>
</tr>
<tr>
<td>+2108</td>
<td>Arg Area Prefix (4 Bytes)</td>
<td></td>
</tr>
<tr>
<td>+2112</td>
<td>Argument Area: Parm 1 Parm 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Local (automatic) Storage</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Saved FPRs Saved ARs</td>
<td></td>
</tr>
<tr>
<td>High Addresses</td>
<td>Stack Pointer (R4) +2048</td>
<td></td>
</tr>
</tbody>
</table>

+2048 +2096 +2104 +2108 +2112

Chapter 3. Using Language Environment debugging facilities 65
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-18 CEECAEYE</td>
<td>CL8/CEECA</td>
</tr>
<tr>
<td>-10 C - -01</td>
<td>Reserved</td>
</tr>
<tr>
<td>000000 CEECAFLAGS</td>
<td>Reserved</td>
</tr>
<tr>
<td>000004</td>
<td>Reserved</td>
</tr>
<tr>
<td>000008 CEECAABOS</td>
<td>Start of Current Storage Segment</td>
</tr>
<tr>
<td>00000C CEECAAEOS</td>
<td>End of Current Storage Segment</td>
</tr>
<tr>
<td>000010</td>
<td>Reserved</td>
</tr>
<tr>
<td>000044 CEECAATORC</td>
<td>POSIX Thread-Level Return Code</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>000074 CEECATOVF</td>
<td>Addr of Stack Overflow Routine</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>000120 CEECAAATTN</td>
<td>Addr of CEL Attention Handler</td>
</tr>
<tr>
<td>000124</td>
<td>Reserved</td>
</tr>
<tr>
<td>00015C CEECAHLEXIT</td>
<td>Flag for User Hook Exit</td>
</tr>
<tr>
<td>0001A8</td>
<td>CEECAHOOKS</td>
</tr>
<tr>
<td>0001F0</td>
<td>Reserved</td>
</tr>
<tr>
<td>0002AC CEECASYSTM</td>
<td>Reserved</td>
</tr>
<tr>
<td>0002B0 CEECALEVE</td>
<td>CAA Level ID</td>
</tr>
<tr>
<td>0002B4 CEECAGETLS</td>
<td>Addr of CEL Library Stack Mgr</td>
</tr>
<tr>
<td>0002B8 CEECAACELV</td>
<td>Addr of CEL LIBVEC</td>
</tr>
<tr>
<td>0002BC CEECAGETS</td>
<td>Addr of CEL Get Stack Stg Rtn</td>
</tr>
<tr>
<td>0002C0 CEECAALBOS</td>
<td>Start of Library Stack Stg Seg</td>
</tr>
<tr>
<td>0002C4 CEECAALEOS</td>
<td>End of Library Stack Stg Seg</td>
</tr>
<tr>
<td>0002C8 CEECAILNAB</td>
<td>Next Available Byte of Lib Stg</td>
</tr>
<tr>
<td>0002CA CEECAADMC</td>
<td>Addr of ESPIE Shunt Routine</td>
</tr>
<tr>
<td>0002D0</td>
<td>CEECAAACD</td>
</tr>
<tr>
<td>0002D4</td>
<td>CEECAAARS</td>
</tr>
<tr>
<td>0002DA</td>
<td>CEECAERR</td>
</tr>
<tr>
<td>0002EC</td>
<td>CEECAGETSX</td>
</tr>
<tr>
<td>0002E0 CEECAADDAA</td>
<td>Addr of the Dummy DSA</td>
</tr>
<tr>
<td>0002E4 CEECAASECTSIZ</td>
<td>Vector Section Size</td>
</tr>
</tbody>
</table>

Figure 10. Common anchor area (Part 1 of 2)
Table 7 contains a list of CAA fields:

<table>
<thead>
<tr>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEECAAPARTSUM</td>
<td>Vector Partial Sum Number</td>
</tr>
<tr>
<td>CEECAASEXPNT</td>
<td>Log of Vector Section Size</td>
</tr>
<tr>
<td>CEECAAEDE8</td>
<td>Addr of the EDB</td>
</tr>
<tr>
<td>CEECAAPCB</td>
<td>Addr of the PCB</td>
</tr>
<tr>
<td>CEECAEYEPRTR</td>
<td>Addr of the CAA Eyecatcher</td>
</tr>
<tr>
<td>CEECAAPTR</td>
<td>Addr of this CAA</td>
</tr>
<tr>
<td>CEECAAGETS1</td>
<td>Stack Overflow for Non-DSA Save Area</td>
</tr>
<tr>
<td>CEECAASHAB</td>
<td>Reserved</td>
</tr>
<tr>
<td>CEECAAPRQCK</td>
<td>Program Interrupt Code for CAADM C</td>
</tr>
<tr>
<td>CEECAFLAG1</td>
<td>Reserved</td>
</tr>
<tr>
<td>CEECAURC</td>
<td>Thread Level Return Code</td>
</tr>
<tr>
<td>CEECAESS</td>
<td>End of Current User Stack</td>
</tr>
<tr>
<td>CEECALESS</td>
<td>End of Current Library Stack</td>
</tr>
<tr>
<td>CEECAOGETS</td>
<td>Overflow from User Stack</td>
</tr>
<tr>
<td>CEECAOGETSL</td>
<td>Overflow from Library Stack</td>
</tr>
<tr>
<td>CEECAAPICICB</td>
<td>Addr of the Preinit Compatibility Control Block</td>
</tr>
<tr>
<td>CEECAARSRV2</td>
<td>Reserved</td>
</tr>
<tr>
<td>CEECAAGOSMR</td>
<td>Reserved</td>
</tr>
<tr>
<td>CEECAaleyov</td>
<td>Addr of z/OS UNIX MVS Library Vector</td>
</tr>
<tr>
<td>CEECA.SIGCTR</td>
<td>SIGSAFE Counter</td>
</tr>
<tr>
<td>CEECA.SIGFLG</td>
<td>SIGSAFE Flags</td>
</tr>
<tr>
<td>CEECA.THID</td>
<td>Thread ID</td>
</tr>
<tr>
<td>CEECAA_DCRENT</td>
<td>Reserved</td>
</tr>
<tr>
<td>CEECAA_DANCHOR</td>
<td>Reserved</td>
</tr>
<tr>
<td>CEECA_CTOC</td>
<td>Reserved</td>
</tr>
<tr>
<td>CEECACICSRSN</td>
<td>CICS Reason Code</td>
</tr>
<tr>
<td>CEECAEMEMBR</td>
<td>Addr of Thread Member List</td>
</tr>
<tr>
<td>CEECA_SIGNAL_STATUS</td>
<td>Addr of Terminating Thread</td>
</tr>
<tr>
<td>CEECA_CEEDLLF</td>
<td>DLL Failure</td>
</tr>
<tr>
<td>CEECA_SAVSTACKASYNC</td>
<td>Zero or saved stack pointer</td>
</tr>
<tr>
<td>CEECA_SAVSTACK</td>
<td>Zero or address of field that is zero or saved stack pointer</td>
</tr>
</tbody>
</table>

Figure 10. Common anchor area (Part 2 of 2)

Table 7 contains a list of CAA fields:

Table 7. List of CAA fields
<table>
<thead>
<tr>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<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>Bit</td>
</tr>
<tr>
<td></td>
<td>0–3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5–7</td>
</tr>
<tr>
<td>CEECAABOS</td>
<td>Start of the current storage segment.</td>
</tr>
<tr>
<td></td>
<td>This field is initially set during thread initialization. It indicates the start of the current stack storage segment. It is altered when the current stack storage segment is changed.</td>
</tr>
<tr>
<td>CEECAAEOS</td>
<td>This field is used to determine if a stack overflow routine must be called when allocating storage from the user stack. Normally, the value of this field will represent the end of the current user stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the user stack. This field is used by function prologs that do not use FASTLINK linkage conventions.</td>
</tr>
<tr>
<td>CEECAATORC</td>
<td>Thread level return code. The thread level return code set by CEESRC callable service.</td>
</tr>
<tr>
<td>CEECAATOVF</td>
<td>Address of stack overflow routine.</td>
</tr>
<tr>
<td>CEECAAATTN</td>
<td>Address of the Language Environment attention handling routine. The address of the Language Environment attention handling routine supports common run-time environment’s polling code convention for attention processing.</td>
</tr>
<tr>
<td>CEECAAHLLEXIT</td>
<td>Address of the Exit List Control Block set by the HLL user exit CEEBINT.</td>
</tr>
<tr>
<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>CAA Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CEECAASYSTM</td>
<td>Underlying operating system. The value indicates the operating system supporting the active environment.</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td><strong>Operating System</strong></td>
</tr>
<tr>
<td>0</td>
<td>Undefined. This value should not appear after Language Environment is initialized.</td>
</tr>
<tr>
<td>1</td>
<td>Unsupported</td>
</tr>
<tr>
<td>3</td>
<td>z/OS</td>
</tr>
<tr>
<td>CEECAAHRDWR</td>
<td>Underlying hardware. This value indicates the type of hardware on which the routine is running.</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td><strong>Hardware</strong></td>
</tr>
<tr>
<td>0</td>
<td>Undefined. This value should not appear after Language Environment is initialized.</td>
</tr>
<tr>
<td>1</td>
<td>Unsupported</td>
</tr>
<tr>
<td>2</td>
<td>System/370™, non-XA</td>
</tr>
<tr>
<td>3</td>
<td>System/370, XA</td>
</tr>
<tr>
<td>4</td>
<td>System/370, ESA</td>
</tr>
<tr>
<td>CEECAASBSYS</td>
<td>Underlying subsystem. This value indicates the subsystem (if any) on which the routine is running.</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td><strong>Subsystem</strong></td>
</tr>
<tr>
<td>0</td>
<td>Undefined. This value should not occur after Language Environment is initialized.</td>
</tr>
<tr>
<td>1</td>
<td>Unsupported</td>
</tr>
<tr>
<td>2</td>
<td>None. The routine is not running under a Language Environment-recognized subsystem.</td>
</tr>
<tr>
<td>3</td>
<td>TSO</td>
</tr>
<tr>
<td>4</td>
<td>IMS™</td>
</tr>
<tr>
<td>5</td>
<td>CICS</td>
</tr>
<tr>
<td>CEECAAFLAG2</td>
<td>CAA Flag 2.</td>
</tr>
<tr>
<td><strong>Bit</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>0</td>
<td>Bimodal addressing is available.</td>
</tr>
<tr>
<td>1</td>
<td>Vector hardware is available.</td>
</tr>
<tr>
<td>2</td>
<td>Thread terminating.</td>
</tr>
<tr>
<td>3</td>
<td>Initial thread</td>
</tr>
<tr>
<td>4</td>
<td>Library trace is active. The TRACE run-time option was set.</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>CEECAA_ENQ_Wait Interruptible. Thread is in an enqueue wait.</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
</tr>
<tr>
<td>CAA Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<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>CEECAA_PM</td>
<td>Image of current program mask.</td>
</tr>
<tr>
<td>CEECAAGETLS</td>
<td>Address of stack overflow for library routines.</td>
</tr>
<tr>
<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>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>CEECAALBOS</td>
<td>Start of the library stack storage segment. This field is initially set during thread initialization. It indicates the start of the library stack storage segment. It is altered when the library stack storage segment is changed.</td>
</tr>
<tr>
<td>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>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>CEECAADMC</td>
<td>Language Environment shunt routine address. Its value is initially set to 0 during thread initialization. If it is nonzero, this is the address of a routine used in specialized exception processing.</td>
</tr>
<tr>
<td>CEECAAAACD</td>
<td>Most recent CAASHAB abend code.</td>
</tr>
<tr>
<td>CEEAAAABCODE</td>
<td>Most recent abend completion code.</td>
</tr>
<tr>
<td>CEECAAAARS</td>
<td>Most recent CAASHAB reason code.</td>
</tr>
<tr>
<td>CEECAAAARSCODE</td>
<td>Most recent abend reason code.</td>
</tr>
<tr>
<td>CEECAERR</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>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>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>
</tbody>
</table>
Table 7. List of CAA fields  (continued)

<table>
<thead>
<tr>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEECAASECTSZ</td>
<td>Vector section size. This field is used by the vector math services.</td>
</tr>
<tr>
<td>CEECAAPARTSUM</td>
<td>Vector partial sum number. This field is used by the vector math services.</td>
</tr>
<tr>
<td>CEECAASSEXPNNT</td>
<td>Log of the vector section size. This field is used by the vector math services.</td>
</tr>
<tr>
<td>CEECAAEEDB</td>
<td>Address of the Language Environment EDB. This field points to the encompassing EDB.</td>
</tr>
<tr>
<td>CEECAAPCB</td>
<td>Address of the Language Environment PCB. This field points to the encompassing PCB.</td>
</tr>
<tr>
<td>CEECAEYEPTTR</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>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>CEECAAGETS1</td>
<td>Non-DSA stack overflow. This field is the address of a stack overflow routine, which cannot guarantee that the current register 13 is pointing at a stack frame. Register 13 must point, at a minimum, to a save area.</td>
</tr>
<tr>
<td>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>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>CEECAAFLAG1</td>
<td>CAA flag bits. The bits are defined as follows:</td>
</tr>
<tr>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>0</td>
<td>CEECAASORT. A call to DFSORT™ is active.</td>
</tr>
<tr>
<td>1</td>
<td>CEECA_USE_OLD_STK. Use the old stack.</td>
</tr>
<tr>
<td>2</td>
<td>CEECA_CICS_EXT_REG. ERTLI CICS extended register interface is in effect</td>
</tr>
<tr>
<td>3–7</td>
<td>Reserved.</td>
</tr>
<tr>
<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>CEECAAES5SS</td>
<td>This field is used to determine if a stack overflow routine must be called when allocating storage from the user stack. Normally, the value of this field will represent the end of the current user stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the user stack. This field is used by function prologs that use FASTLINK linkage conventions.</td>
</tr>
<tr>
<td>CAA Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CEECAALESS</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 use FASTLINK linkage conventions.</td>
</tr>
<tr>
<td>CEECAAOGETS</td>
<td>Overflow from user stack allocations.</td>
</tr>
<tr>
<td>CEECAAOGETLS</td>
<td>Overflow from library stack allocations.</td>
</tr>
<tr>
<td>CEECAARSV1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>CEECAAPICICB</td>
<td>Address of the preinitialization compatibility control block.</td>
</tr>
<tr>
<td>CEECAAOGETSX</td>
<td>User DSA exit from OPLINK.</td>
</tr>
<tr>
<td>CEECAARSV2</td>
<td>Reserved.</td>
</tr>
<tr>
<td>CEECAAGOSMR</td>
<td>Go some more—Used CEEHTRAV multiple.</td>
</tr>
<tr>
<td>CEECAALEOV</td>
<td>This field is the address of the Language Environment library vector for z/OS UNIX support.</td>
</tr>
<tr>
<td>CEECAA_SIGSCTR</td>
<td>SIGSAFE counter.</td>
</tr>
</tbody>
</table>
Table 7. List of CAA fields (continued)

<table>
<thead>
<tr>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEECAATHDID</td>
<td>Thread id. This field is the thread identifier.</td>
</tr>
<tr>
<td>CEECAADCRENT</td>
<td>DCE's read/write static external anchor.</td>
</tr>
<tr>
<td>CEECAADANCHOR</td>
<td>DCE's per-thread anchor.</td>
</tr>
<tr>
<td>CEECAACTOC</td>
<td>TOC anchor for CRENT.</td>
</tr>
<tr>
<td>CEECAACICSRSN</td>
<td>CICS reason code from member language.</td>
</tr>
<tr>
<td>CEECAAMEMBR</td>
<td>Address of thread-level member list.</td>
</tr>
<tr>
<td>CEECAASIGNAL_STATUS</td>
<td>Signal status of the terminating thread member list.</td>
</tr>
<tr>
<td>CEECAASAVSTACK</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>
</tbody>
</table>
Table 7. List of CAA fields (continued)

<table>
<thead>
<tr>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEECAA_SAVSTACK_ASYNC</td>
<td>Zero or address of field that is zero or saved stack pointer. An application that has large sections of code that do not require access to the Language Environment stack but could benefit from having an additional register available can use this field.</td>
</tr>
</tbody>
</table>

For information about the DLL failure control block, CEEDLLF, see z/OS Language Environment Vendor Interfaces.

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 leaw1.h.

Figure 11 on page 75 shows the condition information block.
Figure 11. 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
64 Caused by a program check
128 Caused by an abend

The flags for Status Flag 6, Language Environment actions:

---

1. The signaled-via-CEEOKILL flag is always set with the signaled flag; thus, a signaled condition can have a value of either 2 or 34. (The value is 2 if the signaled condition does not come through CEEOKILL. If it comes through CEEOKILL, its value is 2+32=34.)
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 12 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 89 for the contents of CEEDLLF fields.
Multiple enclave dumps

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 13 on page 80 illustrates the information available in the Language Environment dump and the order of information for multiple enclaves.
Figure 13. 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 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 39.

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

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

**ABPERC**(abcode)

The ABPERC 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:

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 39.

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.

When you are done, you have a generated system dump in a batch run-time environment.

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:

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 39.

   Restriction: In an IMS environment, you can only use CEEUOPT, CEEDOPT, or CEEROPT to change run-time options. CEEUOPT cannot be used by OS/VS COBOL or non-Language Environment assembler.

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.

When you are done, you have a generated system dump in an IMS run-time environment.

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. In order to generate diagnostic information for a CICS run-time environment with a Language Environment U4038 abend, you must create a Language Environment U4039 dump in a batch run-time environment.
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:

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 further details on the `TERMTHDACT` suboptions, see "Generating a Language Environment dump with `TERMTHDACT`" on page 39.

2. Update the transaction dump table with the CICS supplied `CEMT` command:
   ```
   CEMT SET TRD(40XX) SYS ADD
   Result: You will see CEMT output.
   Example:
   TRD(40XX) Sys Loc Max(999) Cur(0000)
   ```

3. Rerun the program.

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

### Steps for generating a Language Environment U4039 abend

If you have a Language Environment U4038 abend, CICS will not generate a system dump. In order to generate diagnostic information, you must create a Language Environment U4039 abend by performing the following steps:

1. Specify `DUMP=YES` in CICS DFHSIT.
2. Relink your program by including `CEEUOPT`.

   **Restriction:** `CEEUOPT` cannot be used by OS/VS COBOL or non-Language Environment assembler.

3. Take `CEECOPT` from `SCEESAMP` and modify the Language Environment run-time options `TERMTHDACT(UAONLY, UATRACE, or UADUMP), ABTERM(ABEND), and TRAP(ON)`.

   **Result:** By setting these run-time options, a Language Environment U4039 abend occurs which generates a system dump.

4. Rerun the program.

   **Note:** In the CICS run-time environment, the `TERMTHDACT` suboption `UAIMM` is processed the same as `UAONLY`.

### 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 command:
       ```
       export _BPXK_MDUMP=filename
       ```
     where `filename` is a fully qualified data set name with DCB information: LRECL=4160, BLKSIZE=4160, and RECFM=FBS.

     **Example:**

Chapter 3. Using Language Environment debugging facilities  83
export _BPXK_MDUMP=hlq.mydump

To write the system dump to an HFS file, issue the command:

export _BPXK_MDUMP=filename

where `filename` is a fully qualified HFS filename.

**Example:**

export _BPXK_MDUMP=/tmp/mydump.dmp

2. Specify Language Environment run-time options:

   export _CEE_RUNOPTS="termthdact(suboption)"

   where `suboption` = UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details regarding the TERMTHDACT suboptions, see "Generating a Language Environment dump with TERMTHDACT" on page 39.

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(hlq,DYNAMIC,TDUMP)"

   where:

   - `suboption` = UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details regarding the TERMTHDACT suboptions, see "Generating a Language Environment dump with TERMTHDACT" on page 39.

   - `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

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

- Ensure that your IPCS job can find the CEEIPCSP member.
  
  IPCS provides an exit control table with imbed statements to enable other products to supply exit control information. The IPCS default table, BLSCECT, normally in the SYS1.PARMLIB library, has the following entry for Language Environment:

    IMBED MEMBER(CEEIPCSP) ENVIRONMENT(IPCS)

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

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

  **Example:**

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

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

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

  EXIT EP(CEEEEANLZ) ANALYZE

Language Environment IPCS Verbexit – LEDATA

Use the LEDATA Verbexit 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

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

Report Type Parameters:

[ SUM ]

[ HEAP | STACK | SM ]

[ HPT(number) ] [ HPTTCB(address) ] [ HPTCELL(address) ]

[ HPTLOC(location) ]

[ CM ]

[ MH ]

[ CEEDUMP ]

[ COMP(value) ]

[ PTBL(value) ]

[ ALL ]

Data Selection Parameters:

[ DETAIL | EXCEPTION ]

Control Block Selection Parameters:

[ CAA(caa-address) ]

[ DSA(dsa-address) ]

[ TCB(tcb-address) ]

[ ASID(address-space-id) ]

[ NTHREADS(value) ]

Parameters

Report type parameters
Use these 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: 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 heapools trace, if available, be formatted. If the value is 0 or *, the trace for every heapools poolid is formatted. If the value is a single number (1-12), the trace for the specific heapools 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 heapool trace table, if available, printing only those entries for a given TCB address (address).

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

HPTLOC(location)
Filters the heapool 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. User 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.

* 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.
CL  Requests a report on C/C++ I/O control blocks.
COBOL  Requests a report on COBOL-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, 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++ and COBOL 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.

Note: 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 107.

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.

Note: For the Summary, CEEDUMP, C/C++, and COBOL 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 CAA and DSA control blocks used as the starting points for formatting.

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

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

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

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

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

Usage Notes: Only hexadecimal characters can be specified as addresses provided in LEDATA parameters. Special characters cause the formatter to fail. Therefore, in order to specify a 64–bit address as a parameter, it must be in the form like 123456789 instead of 1_23456789.

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 14 on page 90 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 CELSAMP in Figure 5 on page 44. “Sections of the Language Environment LEDATA Verbexit formatted output” on page 102 describes the information contained in the formatted output. Ellipses are used to summarize some sections of the dump.

For easy reference, the sections of the following dump are numbered to correspond with the descriptions in “Sections of the Language Environment LEDATA Verbexit formatted output” on page 102.
### Figure 14. Example of formatted output from LEDATA Verbexit (Part 1 of 12)
Figure 14. Example of formatted output from LEDATA Verbexit (Part 2 of 12)
Figure 14. Example of formatted output from LEDATA Verbexit (Part 3 of 12)
Heap Storage Control Blocks

Heap Pools trace available. To display: IP VERBX LEDATA ‘HPT(+) CAA(20915980)’

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENSM</td>
<td>Eye Catcher: ENSM</td>
<td>ST HEAP_ALLOC_FLAG: 00000000</td>
<td>ST HEAP_ALLOC_VAL: 00000000</td>
</tr>
<tr>
<td>+000000</td>
<td>Eye Catcher: ENSM</td>
<td>ST HEAP_ALLOC_FLAG: 00000000</td>
<td>ST HEAP_ALLOC_VAL: 00000000</td>
</tr>
<tr>
<td>+000008</td>
<td>ST HEAP_ALLOC.VAL: 00000000</td>
<td>ST HEAP_FREE_Flag: 00000000</td>
<td></td>
</tr>
<tr>
<td>+000010</td>
<td>ST HEAP_FREE_Flag: 00000000</td>
<td>REPORT_STORAGE: 20914BEC</td>
<td></td>
</tr>
<tr>
<td>+000018</td>
<td>UHEAP: CBD7C82</td>
<td>20FE4018</td>
<td>20FE4018</td>
</tr>
<tr>
<td>+000048</td>
<td>AHEAP: CBD7C82</td>
<td>20FC0000</td>
<td>213E0000</td>
</tr>
<tr>
<td>+000078</td>
<td>BHEAP: CBD7C82</td>
<td>20914B88</td>
<td>20914B88</td>
</tr>
<tr>
<td>+0000A8</td>
<td>ENSM_ADDR_HEAPS: 21366FB8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

STSB: 20914BEC

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>Eye Catcher: STSB</td>
<td>CRHP_REQ: 00000002</td>
<td>DSHP_REQ: 00000001</td>
</tr>
<tr>
<td>+000000</td>
<td>Eye Catcher: STSB</td>
<td>CRHP_REQ: 00000002</td>
<td>DSHP_REQ: 00000001</td>
</tr>
<tr>
<td>+000010</td>
<td>IPR_INIT_SIZE: 00020000</td>
<td>NONIPR_INIT_SIZE: 00020000</td>
<td></td>
</tr>
<tr>
<td>+000014</td>
<td>IPR_INIT_SIZE: 00020000</td>
<td>NONIPR_INIT_SIZE: 00020000</td>
<td></td>
</tr>
<tr>
<td>+00001C</td>
<td>THEAP_MAX_STOR: 00000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Enclave Level Stack Statistics

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKSB: 20914C84</td>
<td>Max Alloc: 000009FB</td>
<td>Curr Alloc: 000008E5</td>
<td></td>
</tr>
<tr>
<td>+000000</td>
<td>Max Alloc: 000009FB</td>
<td>Curr Alloc: 000008E5</td>
<td></td>
</tr>
<tr>
<td>+000008</td>
<td>Larger: 000009FB</td>
<td>GMTAINS: 00000001</td>
<td></td>
</tr>
<tr>
<td>+000010</td>
<td>FREDMAINS: 00000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SKSB: 20914C84

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKSB: 20914C84</td>
<td>Max Alloc: 00000000</td>
<td>Curr Alloc: 00000000</td>
<td></td>
</tr>
<tr>
<td>+000000</td>
<td>Larger: 00000000</td>
<td>GMTAINS: 00000000</td>
<td></td>
</tr>
<tr>
<td>+000010</td>
<td>FREDMAINS: 00000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

User Heap Control Blocks

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPCB: 20914828</td>
<td>Eye Catcher: HPCB</td>
<td>FIRST: 20FE4018</td>
<td>LAST: 20FE4018</td>
</tr>
<tr>
<td>+000000</td>
<td>Eye Catcher: HPCB</td>
<td>FIRST: 20FE4018</td>
<td>LAST: 20FE4018</td>
</tr>
<tr>
<td>+000000</td>
<td>Bytes Alloc: 00000000</td>
<td>Curr Alloc: 00000000</td>
<td></td>
</tr>
<tr>
<td>+000008</td>
<td>Get Req: 00000000</td>
<td>Free Req: 00000000</td>
<td></td>
</tr>
<tr>
<td>+000010</td>
<td>GMTAINS: 00000001</td>
<td>FREDMAINS: 00000000</td>
<td></td>
</tr>
</tbody>
</table>

HPSB: 20914C8C

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPSB: 20914C8C</td>
<td>Bytes Alloc: 00000000</td>
<td>Curr Alloc: 00000000</td>
<td></td>
</tr>
<tr>
<td>+000000</td>
<td>Get Req: 00000000</td>
<td>Free Req: 00000000</td>
<td></td>
</tr>
<tr>
<td>+000010</td>
<td>GMTAINS: 00000001</td>
<td>FREDMAINS: 00000000</td>
<td></td>
</tr>
</tbody>
</table>

HANC: 20FE4018

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HANC: 20FE4018</td>
<td>Eye Catcher: HANC</td>
<td>NEXT: 20914B82</td>
<td>PREV: 20914B82</td>
</tr>
<tr>
<td>+000000</td>
<td>Heap ID: 00000000</td>
<td>Seg ADDR: 20FE4018</td>
<td>Root ADDR: 20FE9240</td>
</tr>
<tr>
<td>+000018</td>
<td>Seg Len: 00000000</td>
<td>Root Len: 000002DB</td>
<td></td>
</tr>
</tbody>
</table>

This is the last heap segment in the current heap.

Free Storage Tree for Heap Segment 20FE4018

<table>
<thead>
<tr>
<th>Node</th>
<th>Address</th>
<th>Length</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>20FE9240</td>
<td>00002DB</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 20FE4018

To display entire segment: IP LIST 20FE4018 LEN(‘X’00000000’) ASID(‘X’003F’)

20FE4038: Allocated storage element, length=00001028. To display: IP LIST 20FE4038 LEN(‘X’00001028’) ASID(‘X’003F’)
20FE4040: 00000005 00000005 20FE5068 20FE5250 20FE52BD 20FE52CA 20FE5307 20FE5344 | ..........& ..........& ..........& |
20FE5060: Allocated storage element, length=00000828. To display: IP LIST 20FE5060 LEN(‘X’00000828’) ASID(‘X’003F’)
20FE5068: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ..........& ..........& ..........& |
20FE5588: Allocated storage element, length=000008CB. To display: IP LIST 20FE5588 LEN(‘X’000008CB’) ASID(‘X’003F’)
20FE5589: 00000003 00000003 C3C43D3D 00000000 00000000 00000000 00000000 00000000 | ..........& ..........& ..........& |
20FE6550: Allocated storage element, length=00000208. To display: IP LIST 20FE6550 LEN(‘X’00000208’) ASID(‘X’003F’)
20FE6558: 00000006 00000006 00000000 00000000 00000000 00000000 | ..........& ..........& ..........& |
20FE8578: Allocated storage element, length=000000CB. To display: IP LIST 20FE8578 LEN(‘X’000000CB’) ASID(‘X’003F’)
20FE8580: 00000001 00000001 00000001 00000001 00000001 00000001 00000001 00000001 | ..........& ..........& ..........& |
20FE9240: Free storage element, length=00002DB. To display: IP LIST 20FE9240 LEN(‘X’00002DB’) ASID(‘X’003F’)

Figure 14. Example of formatted output from LEDATA Verbitext (Part 4 of 12)
Summary of analysis for Heap Segment 20FE4018:
Amounts of identified storage: Free:00002DD8 Allocated:00005208 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.

Anywhere Heap Control Blocks

HPCB: 20914B88 +000000 EYE_CATCHER:HPCB FIRST:20FC0000 LAST:2138E000

HPSB: 20914C3C +000000 BYTES_ALLOC:00371670 CURR_ALLOC:00319DB0
+000008 GET_REQ:00000B58 FREE_REQ:00000B07
+000010 GETMAINS:00000011 FREEMAINS:00000005

HANC: 20FC0000 +000000 EYE_CATCHER:HANC NEXT:20FED000 PREV:20914B88
+00000C HEAPID:20914B88 SEG_ADDR:20FC0000 ROOT_ADDR:20FC3500
+000018 SEG_LEN:00004000 ROOT_LEN:00000B00

Free Storage Tree for Heap Segment 20FC0000

<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>20FC3500</td>
<td>00000B00</td>
<td>00000000</td>
<td>20FC3278</td>
<td>00000000</td>
<td>00000090</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>20FC3278</td>
<td>00000090</td>
<td>20FC3500</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 20FC0000
To display entire segment: IP LIST 20FC0000 LEN('00004000') ASID('003F')

20FC0020: Allocated storage element, length=00000018. To display: IP LIST 20FC0020 LEN('00000018') ASID('003F')

Below Heap Control Blocks

HPCB: 20914B88 +000000 EYE_CATCHER:HPCB FIRST:20914B88 LAST:20914B88

** NO SEGMENTS ALLOCATED **

HPSB: 20914C54 +000000 BYTES_ALLOC:00000000 CURR_ALLOC:00000000
+000008 GET_REQ:00000000 FREE_REQ:00000000
+000010 GETMAINS:00000000 FREEMAINS:00000000

Additional Heap Control Blocks

HPSB: 20914C6C +000000 BYTES_ALLOC:00000000 CURR_ALLOC:00000000
+000008 GET_REQ:00000000 FREE_REQ:00000000
+000010 GETMAINS:00000000 FREEMAINS:00000000

ADHP: 21366F88 +000000 EYE_CATCHER:ADHP NEXT:F0F00000 HEAPID:21366FC4

HPCB: 21366FC4 +000000 EYE_CATCHER:HPCB FIRST:21367000 LAST:21367000

HANC: 21367000 +000000 EYE_CATCHER:HANC NEXT:21367000 PREV:21366FC4
+00000C HEAPID:21366FC4 SEG_ADDR:21367000 ROOT_ADDR:213672CB
+000018 SEG_LEN:00000100 ROOT_LEN:00000038

This is the last heap segment in the current heap.

Free Storage Tree for Heap Segment 21367000

<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>213672CB</td>
<td>00000038</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 21367000

Figure 14. Example of formatted output from LEDATA Verbexit (Part 5 of 12)
To display entire segment: IP LIST 21367000 LEN(X'00000100') ASID(X'003F')

21367020: All located storage element, length=000002A8, To display: IP LIST 21367020 LEN(X'000002A8') ASID(X'003F')

213672CB: Free storage element, length=0000003B, To display: IP LIST 213672CB LEN(X'0000003B') ASID(X'003F')

Summary of analysis for Heap Segment 21367000:
Amounts of identified storage: Free:0000003B  Allocated:000002A8  Total:000002F0
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.


SMCB: 209161B
+000000 EYE_CATCHER:SMCB US_EYE_CATCHER:USTK USFIRST:20FC401B
+00000C USLAST:20FC401B USDBS:20FC401B USDS:20FC401B
+000018 USMB:20FC7D8C USINITSZ:00002000 USINCRSZ:00002000
+000024 USANYBELOW:00000000 USKEEPFREE:00000000 USPOOL:00000000
+000030 USPREALLOC:00000001 USBYTES_ALLOC:00000F28
+000038 US_CURR_ALLOC:00003E50 US_GETMEMS:00000000
+000040 US_FREEMEM:00000000 US_OPLINK:00 LS_THIS_IS:LSTK
+00004C USFIRST:00019000 USLAST:00019000 USDBS:00019000
+000058 USDS:0001A000 USMB:0000A000 USINITSZ:00001000
+000064 USINCRSZ:00010000 USANYBELOW:00000000
+00006C USKEEPFREE:00000001 USPOOL:00000000 USPREALLOC:00000000
+000078 USBYTES_ALLOC:00000000 LS_GETMEMS:00000000
+000080 LS_FREEMEM:00000000 LS_OPLINK:00 LS_THIS_IS:LSTK
+00008C RSDBS:20FC4000 RSDB:20FC401B RSIZE:00000000
+000098 RSACTIVE:00000001 SA_REG0:20FC7E68
+0000A0 SA_REG0:20FC7CDB SA_REG0:20FC4648
+0000A8 SA_REG0:20FC4648 SA_REG0:20FC499C
+0000B0 SA_REG0:20FC512C SA_REG0:00000000
+0000B8 SA_REG0:20FC92A8 SA_REG0:0AFC83C2
+0000C0 SA_REG0:20FC8CA7 SA_REG0:20FC5CA7
+0000CB SA_REG0:20FC47C0 SA_REG0:20FC159B0
+0000D0 SA_REG0:20FC4CA8 SA_REG0:20FC4A2
+0000D8 SA_REG0:00000000
+0000DC SAVEREG_XINIT:00000000 00000000 00000000 00000000
+0000EC CEENVTSI:80012D0 ST_DSA_ALLOC_FLAG:00000000
+0000FC ST_DSA_ALLOC_FLAG:00000000 ALLOCSEG:00000000
+000100 LOCAL_GETMEM:00000000 LOCAL_ALLOC:FFFFF000
+00010C LOCAL_GETMEMS:00000000 LOCAL_FREEMEM:00000000
+00015C MOREFLAGS:00000000 DS_THIS_IS:.... DFIRST:20916278
+000168 DSTART:2091627B DSFIRST:2091627B DSINITSZ:00000000
+00017C DSINITSZ:00000000 DSARGSZ:00000000
+000184 DSKEEPFREE:00000000 DSPAN:00000000 DSFREEALLOC:00000000
+000190 DSBYTES_ALLOC:00000000 DS_CURR_ALLOC:00000000
+000198 DS_FREEMEM:00000000 DS_FREEMEM:00000000
+0001A0 DS_FLAGS:00000000

DSA backchain

DSA: 20FC7C6B
+000000 FLAGS:0000 MEMP:0000 BKC:20FC4CA8 FWC:20FC7ECB
+00000C R14:0AFA4F4E R15:0AFAE728 R0:20992B40
+000018 R1:20FC7585 R2:20FC4F00 R3:0A917778
+000024 R4:20FC7E78 R5:20FC4030 R6:00000000
+000030 R7:20FC7E8B R8:00000001 R9:FFFFFFFF
+000038 R10:00000000 R11:0AFA4B8 R12:20915980
+000044 LWS:00000000 NAB:20FC7E68 PMAB:00000000
+000048 RENT:00000000 CILC:00000000 MODE:00000000
+000078 RMR:00000000

Contents of DSA at location 20FC7CB:

+00000000 00000000 20FC4CA8 20FC7CEB A07A4F4E A07A4E728 20992B40 20FC4F00 20FC7E8B 
+000002A0 0A917778 20FC7E78 20FC4030 00000000 20FC7E68 00000001 FFFFFFFC 00000000
+00000400 A07A4E88 20915980 00000000 20FC7E68 00000000 00000000 00000000 00000000
+00000600 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
+00000800 20FC512C 20FC5244 20FC515C 20FC5188 20FC7E8C 20FC7E7C 20FC7E68 
+000012A8 .........d...h.........0..y

DSA: 20FC4CA8
+000000 FLAGS:0000 MEMP:0000 BKC:20FC4208 FWC:20FC7C6B
+00000C R14:0A9CB85B R15:20904C70 R0:00000007
+000018 R1:20FC7150 R2:20914648 R3:20914198
+000024 R4:20FC949C R5:20FC512C R6:00000000
+000030 R7:20FC92A8 R8:0A9CB3C2 R9:20FC6CA6
+000038 R10:20FC5CA7 R11:A09C4870 R12:20915980
+000044 LWS:00000000 NAB:20FC7C6B PMAB:20FC663A
+000048 RENT:213D72AC CILC:2090EFF0 MODE:0000100B
+000078 RMR:20FC6630

Figure 14. Example of formatted output from LEDATA Verexit (Part 6 of 12)
Figure 14. Example of formatted output from LEDATA Verbexit (Part 7 of 12)
Figure 14. Example of formatted output from LEDATA Verbexit (Part 8 of 12)
Figure 14. Example of formatted output from LEDATA Verbexit (Part 9 of 12)
Figure 14. Example of formatted output from LEDATA Verbexit (Part 10 of 12)
Figure 14. Example of formatted output from LEDATA Verbexit (Part 11 of 12)
Figure 14. Example of formatted output from LEDATA Verbexit (Part 12 of 12)
Sections of the Language Environment LEDATA Verbexit formatted output
The sections of the output listed here appear independently of the Language Environment-conforming languages used.

These sections are included when the SUMMARY parameter is specified on the LEDATA invocation.

[1] Summary Header
The summary header section contains:
• Address of Thread control block (TCB)
• Release number
• Address Space ID (ASID)

[2] Active Members List
This list of active members is extracted from the enclave member list (MEML).

[3] CEECAA
This section formats the contents of the Language Environment common anchor area (CAA). See "The Common Anchor Area" on page 65 for a description of the fields in the CAA.

[4] CEEDLLF
This section 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
This section formats the contents of the Language Environment process control block (PCB), and the process level member list.

[6] CEERCB
This section formats the contents of the Language Environment region control block (RCB).

[7] CEEEDB
This section formats the contents of the Language Environment enclave data block (EDB), and the enclave level member list.

[8] PMCB
This section formats the contents of the Language Environment program management control block (PMCB).

[9] Run-Time Options
This section 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.

This section 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 107.


This section is included when the STACK or SM parameter is specified on the LEDATA invocation.

This section formats:
• Storage management control block (SMCB)
• Chain of dynamic save areas (DSA)
  See “The upward-growing (non-XPLINK) stack frame section” on page 63 or “The 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.

This section 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 “The condition information block” on page 74 for a description of fields in these control blocks.


This section is included when the MH parameter is specified on the LEDATA invocation.


These sections are included when the CEEDUMP 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 “Report type parameters” on page 86.

[14] Enclave Identifier

This statement names the enclave for which information is provided.

[15] Information for thread

This section shows the system identifier for the thread. Each thread has a unique identifier.

[16] Registers and PSW

This section 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.

[17] Traceback

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

- DSA number
  A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second part of the traceback.
- Entry
  For COBOL, Fortran, and PL/I routines, this is the entry point name. For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, the string "** NoName **" will appear.
- Entry point offset
- Statement number
  This field contains no Language Environment data.
- Load module
- Program unit
  The primary entry point of the external procedure. For COBOL programs, this is the PROGRAM-ID name. For C, Fortran, and PL/I routines, this is the compile unit name. For Language Environment-conforming assemblers, this is the EPNAME = value on the CEEPPA macro.
- Service level
  The latest service level applied to the compile unit (for example, for IBM products, it would be the PTF number).
- 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 including:
  - 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.

[18] Control Blocks Associated with the Thread

This section lists the contents of the thread synchronization queue element (SQEL).

[19] Enclave Control Blocks

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.

[20] Language Environment Trace Table

If the TRACE run-time option was set to ON, this section shows the contents of the Language Environment trace table.

[21] Process Control Blocks

If the POSIX run-time option was set to ON, this section lists the contents of the process level latch table.

[22] Preinitialization Information

This section is included when the PTBL parameter is specified on the LEDATA invocation.
This section formats information related to preinitialization. See PTBL LEDATA output for more information. If the preinitialization service CEEPIPI was not used to initialize this environment, the message: No PIPLICB 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 15 illustrates the output produced when the Verbexit LEDATA command is invoked with the PTBL parameter.

```
PTBL(CURRENT)
********************************************************************************
LANGUAGE ENVIRONMENT DATA
********************************************************************************

Language Environment Product 04 V01 R09.00

PreInitialization Programming Interface Trace Data
CEEPIPI Environment Table Entry and Trace Entry :
Active CEEPIPI Environment ( Address 20905CB0 )
Eyecatcher : CEEXIPTB
TCB address : 00006E88

CEEPIPI 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 ( 20905BB8 )
+000000 20905BB8 A0910620 20919B30 80000000 00000000 00000000 00000000 00000000 |.j...j..........................|
+000020 20905DB0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |.............j.......jpQ.....j..|
+000040 20905DF8 A0910530 000004D0 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 15. 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 16 on page 108 illustrates the output produced by specifying the HEAP option. "Heap report sections of the LEDATA output" on page 111 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. 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.

CEEPIPI Trace Table Entries:
Call Type = INIT_MAIN
PIPI Driver Address = A090068A
Load Service Return Code = 0
Load Service Reason Code = 0
Most Recent Return Code = 0
Most Recent Reason Code = 0
An ABEND will be issued if storage can not be obtained
PreInit Environment will not allow EXEC CICS commands
Service RC = 0 : A new environment was initialized.

Call Type = ADD_ENTRY
Routine Table Index = 1
Routine Name = ISJPPCA1
Routine Address = A0FCC548
Load Service Return Code = 0
Load Service Reason Code = 3
Service RC = 0 : The routine was added to the PreInit table.

Call Type = IDENTIFY_ENTRY
Routine Table Index = 1
Routine Programming Language = C/C++
Service RC = 0 : The programming language has been returned.

Call Type = CALL_MAIN
Routine Table Index = 1
Enclave Return Code = 0
Enclave Reason Code = 0
Routine Feedback Code = 0000000000000000
Service RC = 0 : The environment was activated and the routine called.

Call Type = DELETE_ENTRY
Routine Table Index = 1
Routine Name = ISJPPCA1
Routine Address = A0FCC548
Service RC = 0 : The routine was deleted from the PreInit table.

Call Type = CALL_MAIN
Routine Table Index = 0
Enclave Return Code = 0
Enclave Reason Code = 0
Routine Feedback Code = 0000000000000000
Service RC = 0 : The environment was activated and the routine called.

Exiting Language Environment Data

Figure 15. Example of formatted PTBL output from LEDATA Verbexit (Part 2 of 2)
FREE STORAGE TREE FOR HEAP SEGMENT 25995000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Node</th>
<th>Node</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>259950B0</td>
<td>000007F50</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MAP OF HEAP SEGMENT 25995000

To display entire segment: IP LIST 25995000 LEN(X'00008000') ASID(X'0021')

FREE STORAGE TREE FOR HEAP SEGMENT 25995000

The free storage tree is empty.

Figure 16. 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('00F00028') ASID('0021')

24A91020: Allocated storage element, length=00F00008. To display: IP LIST 24A91020 LEN('00F00008') ASID('0021')
24A91028: B035F8D8 82C00081 24A8E080 00000000 00000001 94B19995 40404040 ...Q...a.............main |

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

HANC: 259AC000
EYE_CATCHER:HANC NEXT:259AF000 PREV:2599D000
SEG_LEN:00002000
HEAPID:00014D78
SEG_ADDR:259AC000
EYE_CATCHER:HANC
PREV:2599D000
SEG_LEN:00000C30

Free Storage Tree for Heap Segment 259AC000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Length</th>
<th>Node</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>259AC020 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>259AC048 00000038 259AC020 00000000 00000000 00000000 00000000 00000000 00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Map of Heap Segment 259AC000

To display entire segment: IP LIST 259AC000 LEN('00000200') ASID('0021')

259AC020: Free storage element, length=000000C30. To display: IP LIST 259AC020 LEN('000000C30') ASID('0021')

259AC530: Allocated storage element, length=000000728. To display: IP LIST 259AC530 LEN('000000728') ASID('0021')
259AC538: D033E340 071C0001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |

Summary of analysis for Heap Segment 259AC000:
Amounts of identified storage: Free:00000000 Allocated:00000000 Total:00000000
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.

...
This is the last heap segment in the current heap.

Free Storage Tree for Heap Segment 00044000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Length</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>00001C78</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 00044000

To display entire segment: IP LIST 00044000 LEN(X'00002000') ASID(X'0021')

00044020: Allocated storage element, length=00000048. To display: IP LIST 00044020 LEN(X'00002000') ASID(X'0021')

00044218: Allocated storage element, length=00000048. To display: IP LIST 00044218 LEN(X'00000048') ASID(X'0021')

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:259ADC08 HEAPID:259B24B4

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

ADHP: 259ADC08
  +000000 EYE_CATCHER:ADHP NEXT:F0F00000 HEAPID:259ADC14

HPCB: 259ADC14
  +000000 EYE_CATCHER:HPCB FIRST:259AE000 LAST:259AE000

HANC: 259AE000
  +000000 EYE_CATCHER:HANC NEXT:259ADC14 PREV:259ADC14

Figure 16. 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.


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


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:
- 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](https://www.ibm.com/support/knowledgecenter/SSD750_10.2.0/com.ibm.zos.v1r2.ce.doc/ce00np0002.html).

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 17
on page 114 illustrates the details of heappool report. "Heappool report sections of
the LEDATA output" on page 118 describes the information contained in the
formatted output.
Figure 17. Example formatted detailed heappool report from LEDATA Verbexit (Part 1 of 4)
Chapter 3. Using Language Environment debugging facilities

| EXTENT: 25E4802B |
| +000000 EYE_CATCHER:EX31 NEXT_EXTENT:25C42040 |
To display entire pool extent: IP LIST 25E4802B LEN('X'0000010B') ASID('X'0020')

| 25E48030: Free storage cell. To display: IP LIST 25E48030 LEN('X'00000040B') ASID('X'0020') |

### EXTENT: 25C42040

+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000 |
To display entire pool extent: IP LIST 25C42040 LEN('X'0000010B') ASID('X'0020')

| 25C42048: Allocated storage cell. To display: IP LIST 25C42048 LEN('X'00000040B') ASID('X'0020') |


| 25C42450: Allocated storage cell. To display: IP LIST 25C42450 LEN('X'00000040B') ASID('X'0020') |

| 25C42528: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |

| 25C42858: Free storage cell. To display: IP LIST 25C42858 LEN('X'00000040B') ASID('X'0020') |

| 25C42C60: Free storage cell. To display: IP LIST 25C42C60 LEN('X'00000040B') ASID('X'0020') |

### [1] 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: 3 Free: 3 Allocated: 2 Total Used: 8
- No errors were found while processing this Pool.

#### Data for pool 5.2:

| POOLDATA: 25C1F300 |
| +000000 POOL_INDEX:00000006 INPUT_CELL_SIZE:00000408 |


| EXTENT: 25C463F8 |
| +000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000 |
To display entire pool extent: IP LIST 25C463F8 LEN('X'000010B') ASID('X'0020')

| 25C46400: Free storage cell. To display: IP LIST 25C46400 LEN('X'00000040B') ASID('X'0020') |

| 25C46808: Free storage cell. To display: IP LIST 25C46808 LEN('X'00000040B') ASID('X'0020') |

| 25C46C10: Free storage cell. To display: IP LIST 25C46C10 LEN('X'00000040B') ASID('X'0020') |

### [1] Verifying free chain for pool: 5.2...

No errors were found while processing free chain.

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

#### Data for pool 5.3:

| POOLDATA: 25C1F400 |
| +000000 POOL_INDEX:00000007 INPUT_CELL_SIZE:00000408 |


| EXTENT: 25E49058 |
| +000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000 |
To display entire pool extent: IP LIST 25E49058 LEN('X'000010B') ASID('X'0020')

| 25E49060: Free storage cell. To display: IP LIST 25E49060 LEN('X'00000040B') ASID('X'0020') |

| 25E49468: Free storage cell. To display: IP LIST 25E49468 LEN('X'00000040B') ASID('X'0020') |

| 25E49870: Free storage cell. To display: IP LIST 25E49870 LEN('X'00000040B') ASID('X'0020') |

### [1] Verifying free chain for pool: 5.3...

No errors were found while processing free chain.

Summary of analysis for Pool 5.3:
- Number of cells: Unused: 1 Free: 3 Allocated: 0 Total Used: 4
- No errors were found while processing this Pool.

---

Figure 17. Example formatted detailed heap pool report from LEDATA Verbexit (Part 3 of 4)
Figure 17. Example formatted detailed heappool report from LEDATA Verbexit (Part 4 of 4)
Heappool report sections of the LEDATA output

The Heappool report provides the information regarding 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.

[1]Free Chain Validation

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.


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 in order to identify the reason for the storage allocation.

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.
HPT(3)
****************************************************************************
LANGUAGE ENVIRONMENT DATA
****************************************************************************
Language Environment Product 04 V01 R10.00

[1] HEAPPOOLS Trace Table


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

[4] CALL NAME          CALL ADDRESS  CALL OFFSET
GetStorage::"GetStorage()  25E53360  00000088
foo8() 25E53590  000000B6
foo7() 25E53678  0000005A
foo6() 25E536F0  0000005A
foo5() 25E5376B  0000005A
foo4() 25E537EB  0000005A
foo3() 25E5385B  0000005A
foo2() 25E538D0  0000005A
foo1() 25E5394B  0000005A
thread 25E53A50  00000000

Figure 18. Example formatted detailed HEAPPOOLS trace report from LEDATA Verbxit (Part 1 of 4)
<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 14:10:22.614087</th>
<th>Type: FREE Cell Address: 25E91848 CpuId: 01 Tcb: 008AFFC0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL NAME CALL ADDRESS CALL OFFSET</td>
<td></td>
</tr>
<tr>
<td>GetStorage::&quot;GetStorage()&quot; 25E53360 00000088</td>
<td></td>
</tr>
<tr>
<td>foo9() 25E53430 00000086</td>
<td></td>
</tr>
<tr>
<td>foo8() 25E53598 0000009A</td>
<td></td>
</tr>
<tr>
<td>foo7() 25E53678 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo6() 25E536F0 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo5() 25E53768 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo4() 25E537E0 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo3() 25E5385B 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo2() 25E5380D 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo1() 25E53948 00000000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 14:10:22.614034</th>
<th>Type: FREE Cell Address: 25E918D0 CpuId: 01 Tcb: 008AF46F</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL NAME CALL ADDRESS CALL OFFSET</td>
<td></td>
</tr>
<tr>
<td>GetStorage::&quot;GetStorage()&quot; 25E53360 00000088</td>
<td></td>
</tr>
<tr>
<td>foo9() 25E53430 00000086</td>
<td></td>
</tr>
<tr>
<td>foo8() 25E53598 0000009A</td>
<td></td>
</tr>
<tr>
<td>foo7() 25E53678 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo6() 25E536F0 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo5() 25E53768 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo4() 25E537E0 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo3() 25E5385B 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo2() 25E5380D 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo1() 25E53948 00000000</td>
<td></td>
</tr>
<tr>
<td>thread 25E53A50 00000000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 14:10:22.614032</th>
<th>Type: FREE Cell Address: 25E91C58 CpuId: 01 Tcb: 008AF6A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL NAME CALL ADDRESS CALL OFFSET</td>
<td></td>
</tr>
<tr>
<td>GetStorage::&quot;GetStorage()&quot; 25E53360 00000088</td>
<td></td>
</tr>
<tr>
<td>foo9() 25E53430 00000086</td>
<td></td>
</tr>
<tr>
<td>foo8() 25E53598 0000009A</td>
<td></td>
</tr>
<tr>
<td>foo7() 25E53678 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo6() 25E536F0 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo5() 25E53768 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo4() 25E537E0 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo3() 25E5385B 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo2() 25E5380D 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo1() 25E53948 00000000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp: 2008/03/14 14:10:22.614030</th>
<th>Type: GET Cell Address: 25E91C58 CpuId: 01 Tcb: 008AF6A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL NAME CALL ADDRESS CALL OFFSET</td>
<td></td>
</tr>
<tr>
<td>GetStorage::GetStorage(int) 25E53298 0000008C</td>
<td></td>
</tr>
<tr>
<td>foo9() 25E53430 00000086</td>
<td></td>
</tr>
<tr>
<td>foo8() 25E53598 0000009A</td>
<td></td>
</tr>
<tr>
<td>foo7() 25E53678 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo6() 25E536F0 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo5() 25E53768 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo4() 25E537E0 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo3() 25E5385B 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo2() 25E5380D 0000005A</td>
<td></td>
</tr>
<tr>
<td>foo1() 25E53948 00000000</td>
<td></td>
</tr>
</tbody>
</table>

Figure 18. Example formatted detailed HEAPPOOLS trace report from LEDATA Verbexit (Part 2 of 4)
Figure 18. Example formatted detailed HEAPPOOLS trace report from LEDATA Verbexit (Part 3 of 4)
[1] Trace Header

HEAPPOOLS trace header information.

[2] Pool Information

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.

[3] Timestamp

The time this trace entry was taken.

Note: The trace entries are formatted in reverse order (most recent trace entry first).

[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 ALL parameter is specified and C/C++ is active in the dump. Figure 19 on page 124 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 CELSAMP. Figure 5 on page 44. "C/C++-specific sections of the LEDATA output" on page 145 describes the information contained in the formatted output. Ellipses are used to summarize some sections of the dump.

For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 1 of 22)
Chapter 3. Using Language Environment debugging facilities

Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 2 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 3 of 22)
Exiting CRTL Environment Data

********************************************************************************

**CRTL I/O CONTROL BLOCKS**

**********CRTL I/O CONTROL BLOCKS**********

**CIO:** 20C0E098
+000000 EYE:CIO SIZE:00000090 PTR:00000000 FLG1:09
+000000 FLG2:E0 FLG3:00 FLG4:00 DUMMYF:20C0E128
+000014 EDCZ24:00000000 FCBSSTART:2135A40 DUMMYF:20C0E148
+000020 MFCBSTART:2135A470 IOANYLIST:2135A000
+000028 IOBELOWLIST:00140000 FCBDDLIST:00000000
+000030 PERRORBUF:20C0F6F0 TMPCOUNTER:00000000
+000038 TEMPMEM:00000000 PROMPTBUF:00000000 1024:00012200
+000044 IOEXITS:000127CB TERMINALCHAIN:00000000
+00004C VANCHOR:00000000 ENOWP24:20F484A0
+000058 MAXNUMDESCRPS:00000000 DESCARRAY:00000000
+000064 TEMPFILENUM:00000000 CSS:00000000 DUMMY_NAME:........
+000074 HOSTNAME_CACHE:00000000 HOSTADDR_CACHE:00000000
+00007C 1031:20E62508 LAST_FD_CLOSE:00000000 00000000
+000080 10GET64:213244E8 IOFREE64:21323D80

File name: /u/charum/b235/in.txt

Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 4 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 5 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 6 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 7 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 8 of 22)
### DCB:

- **DCBEID:** DCBE
- **DCBLEN:** 003B
- **RESERVED:** 0000
- **DCBECB:** 00129F0
- **DCBERELA:** 00000000
- **DCBEFLG1:** 00
- **DCBEFLG2:** 00
- **DCBEFLG3:** 00
- **DCBESIZE:** 00000000
- **DCBEEDC:** 20E6261A
- **DCBESYN1:** 20E626263
- **MULTSDN:** 00

### JFCB:

- **JFCSI:** 00012908
- **JFCBDSNM:** CHARUM.D.E
- **JFCBELNM:** JFCS
- **JFCBTSDM:** 80
- **JFCBDSCB:** 003E15
- **JFCBVLSQ:** 0000
- **JFCBIND1:** 00
- **JFCBIND2:** 40
- **JFCBUFNO:** 00
- **JFCDSRG1:** 00
- **JFCDSRG2:** 00
- **JFCRECFM:** 00
- **JFCBLKSI:** 0000
- **JFCLRECL:** 0000
- **JFCNCP:** 00
- **JFCBNVOL:** 01
- **JFCBNVOLS:** SL621D
- **JFCBEXTL:** 00
- **JFCBEXAD:** 0009FF
- **JFCBVLCT:** 01

### MBUF:

- **NEXTMBUF:** 000129C0
- **BUFFER:** 2135AFE0
- **CHECKRESULT:** 00000000
- **BLKSIZE:** 00000050

### FFIL:

- **MARKER1:** AFCB
- **FILE:** 00000000
- **__FP:** 2135B0A0
- **MARKER2:** AFCBAFCB
- **FF_FLAGS:** 01000000
- **FCBMUTEX:** 00000000
- **THREADID:** 00000000

### FCB:

- **BUFPTR:** 00000000
- **COUNTIN:** 00000000
- **COUNTOUT:** 00000000
- **READFUNC:** 2135B170
- **WRITEFUNC:** 2135B190
- **FLAGS1:** 0000
- **DEPTH:** 0000
- **NAME:** 2135B25C
- _LENGTH_: 0000000A
- **_BUFSIZE:** 00000044
- **MEMBER:** ........
- **NEXT:** 2135B370
- **PREV:** 2135AD30
- **PARENT:** 2135B0A0
- **CHILD:** 00000000
- **DDNAME:** SYS00013
- **FD:** FFFFFFFF
- **DEVTYPE:** 00
- **FCBTYPE:** 0020
- **FSCE:** 2135B1D4
- **UNGETBUF:** 2135B1D4
- **REPOS:** 20F7CAF0
- **GETPOS:** 20F7CEC0
- **CLOSE:** 20F77BE8
- **FLUSH:** 20F7CA78
- **UTILITY:** 213126A0
- **USERBUF:** 00000000
- **LRECL:** 00000404
- **BLKSIZE:** 00000408
- **REALBUFPTR:** 00000000
- **UNGETCOUNT:** 00000000
- **BUFSIZE:** 00000408
- **BUF:** 00000000
- **CURSOR:** 00000000
- **ENDOFDATA:** 00000000
- **SAVEDBUF:** 00000000
- **REALCOUNTIN:** 00000000
- **REALCOUNTOUT:** 00000000
- **POSMAJOR:** 00000000
- **SAVEMAJOR:** 00000000
- **STATE:** 0000
- **SAVESTATE:** 0000
- **EXITFTELL:** 00000000
- **EXITUNGETC:** 20F481E8
- **DBCSTART:** 00000000
- **UTILITYAREA:** 00000000
- **FLAGS2:** 02120020
- **20441100
- **DBCSSTATE:** 0000
- **FCB_CPCB:** 20C0DEA8
- **READGLUE:** 58FF0008
- **07FF0000
- **READ:** 20F48580
- **RADDR_WSA:** 00000000
- **_GETFN:** 00000000
- **RDLL_INDEX:** 00000000
- **RCEESG003:** 00000000
- **RWSA:** 00000000
- **WRITEGLUE:** 58FF0008
- **07FF0000
- **WRITE:** 20F7D260
- **WADDR_WSA:** 00000000
- **_GETFN:** 00000000
- **WDLL_INDEX:** 00000000
- **WCEESG003:** 00000000
- **WWSA:** 00000000

### OSVF:

- **OSVF_EYE:** OSVF
- **READ:** 20F48580
- **WRITE:** 20F7ADD8
- **REPOS:** 20F786B0
- **GETPOS:** 20F7A5C8
- **CLOSE:** 00000000
- **FLUSH:** 20F77F38
- **UTILITY:** 213126A0
- **EXITFTELL:** 00000000
- **EXITUNGETC:** 20F481E8
- **OSIOBLK:** 2135B2A8
- **SAVECURSOR:** 00000000
- **NEWLINEPTR:** 00000000
- **CURBLKSIZE:** 00000000
- **LASTBLKSIZE:** 00000000
- **HIGHMAJOR:** 00000000
- **MAXFTELLBLK:** 001FFFFF
- **RECBITS:** 0000000B
- **FLAGS:** 00000000
- **RECCOUNTOUT1:** 00000000
- **BLKCOUNTOUT1:** 00000000
- **SAVECOUNTOUT:** 00000000
- **SDWCODE:** 00
- **SAVECHAR:** ..
- **LASTTTRBLT:** 00000000
- **LASTWRTSIZE:** 00000000
- **READERCOUNT:** 00000000
- **UNWRITTENDATA:** 00000000
- **MINRECL:** 00000000

---

**Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 9 of 22)**
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 10 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 11 of 22)
Figure 19. Example formatted C++ output from LEDATA Verbexit (Part 12 of 22)

Chapter 3. Using Language Environment debugging facilities
**Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 13 of 22)**

![Image of formatted C/C++ output from LEDATA Verbexit](image-url)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 14 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 15 of 22)
<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0000B4</td>
<td>CURSOR:00000000 ENDODATA:00000000 SAVEDBUF:00000000</td>
</tr>
<tr>
<td>+000090</td>
<td>REALCOUTN:00000000 REALCOUTOUT:00000000</td>
</tr>
<tr>
<td>+00009B</td>
<td>PSMAJOR:00000000 SAVEMAJOR:00000000</td>
</tr>
<tr>
<td>+0000A0</td>
<td>PEMAJOR:00000000 SAVEMINOR:00000000 STATE:0000</td>
</tr>
<tr>
<td>+0000AA</td>
<td>SAVESTATE:0000 EXITFTELL:00000000 EXITUNGETC:20F4B1EB</td>
</tr>
<tr>
<td>+0000B4</td>
<td>DBSTART:00000000 UTILITAAREA:00000000</td>
</tr>
<tr>
<td>+0000BC</td>
<td>INTERCEPT:00000000 FLAGS2:02220020 60440100</td>
</tr>
<tr>
<td>+0000CB</td>
<td>DBSSTATE:0000 FCB_CPCB:20C0DEA8</td>
</tr>
<tr>
<td>+0000D0</td>
<td>READGLUE:58FF0008 07FF0000 READ:20F4858B</td>
</tr>
<tr>
<td>+0000EB</td>
<td>RCEESG03:00000000 RWSA:00000000</td>
</tr>
<tr>
<td>+0000F0</td>
<td>WRITEGLUE:58FF0008 07FF0000 WRITE:20F5D4E8</td>
</tr>
<tr>
<td>+0000FC</td>
<td>WADWR_WS:00000000 GETFN:00000000 WDLL_INDEX:00000000</td>
</tr>
<tr>
<td>+0010B</td>
<td>WCEESG03:00000000 WWSA:00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSUT:</td>
<td>2135BFE4</td>
</tr>
<tr>
<td>+000000</td>
<td>OSUT_EYE:OSUT READ:20F4858B WRITE:20FBE8B0</td>
</tr>
<tr>
<td>+0000C</td>
<td>REPOS:20FBCAD0 GETPOS:20FD6EC8 CLOSE:00000000</td>
</tr>
<tr>
<td>+0001B</td>
<td>FLUSH:20FBC5C0 UTILITY:20FBE70B EXITFTELL:00000000</td>
</tr>
<tr>
<td>+00024</td>
<td>EXITUNGETC:20F4B1EB OSIOBLK:2135C0B8</td>
</tr>
<tr>
<td>+0002C</td>
<td>NEWLINEPTR:00000000 MAXFTELLBLK:0007FFFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSIO:</td>
<td>2135C0B8</td>
</tr>
<tr>
<td>+000000</td>
<td>OSIO_EYE:OSIO DCB:00014020 DCBRU:00000000</td>
</tr>
<tr>
<td>+0000C</td>
<td>JFCB:00014088 CURRRBUF:00000000 MBUF:00000000</td>
</tr>
<tr>
<td>+0001B</td>
<td>READER:00000000 CURRELNUM:FFFFF FF CURRELNUM:FFFFF FF</td>
</tr>
<tr>
<td>+00020</td>
<td>LASTRELNUM:FFFFFFFF BLSKPERTRK:0000 OSIO_ACCESS_METHOD:01</td>
</tr>
<tr>
<td>+00027</td>
<td>NOSEEK_TO_SEEK:00 FIRSTPOS:00000010</td>
</tr>
<tr>
<td>+0002C</td>
<td>LASTPOS:00000010 NEWPOS:00000000 READFUNCNUM:00000002</td>
</tr>
<tr>
<td>+0003B</td>
<td>WRITEFUNCNUM:0000000000 FCB:2135BEB0 PARENT:2135C0B8</td>
</tr>
<tr>
<td>+00044</td>
<td>FLAGS1:81020000 DCBERRU:00000000 DCBEW:2135C120</td>
</tr>
<tr>
<td>+0005B</td>
<td>OSIO_HIGHVOL:0000 APPENDEDLASTVOLSEQ:0000</td>
</tr>
<tr>
<td>+0005C</td>
<td>OSIO_JFCB:00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCB:</td>
<td>00014020</td>
</tr>
<tr>
<td>+000000</td>
<td>DCBLRELAD:2135C120 DCBDAD:00000000 10000000</td>
</tr>
<tr>
<td>+000014</td>
<td>DCBUPNO:00 DCBSG:40 DCBEPAD:00000000 DCBEFCM:CO</td>
</tr>
<tr>
<td>+000025</td>
<td>DCBEOLSA:001270 DCBDOWNAM:......c. DCBMACR1:65</td>
</tr>
<tr>
<td>+000033</td>
<td>DCBMCR2:00 DCBSYNAD:000000 DCBLSK1:1800 DCBNCP:01</td>
</tr>
<tr>
<td>+000052</td>
<td>DCBLRECL:0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCBE:</td>
<td>2135C120</td>
</tr>
<tr>
<td>+000000</td>
<td>DCBEID:DCBE DCBELEN:0000 RESERVE:0000</td>
</tr>
<tr>
<td>+000008</td>
<td>DCBEDCB:00014020 DCBERELA:00000000 DCBEFLG1:CO</td>
</tr>
<tr>
<td>+000011</td>
<td>DCBEFLG2:00 DCBENSTR:0000 DCBEFLG3:00</td>
</tr>
<tr>
<td>+000024</td>
<td>DCBESIZE:00000000 DCBEEEODA:20E6269A</td>
</tr>
<tr>
<td>+00002C</td>
<td>DCBESYN:20E6263C MULTSDN:00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFCB:</td>
<td>00014088</td>
</tr>
<tr>
<td>+000000</td>
<td>JFCBDSMN:CHARUM.J.K</td>
</tr>
<tr>
<td>+00002C</td>
<td>JFCBELNM: JFCBDTSMD:80 JFCBECSDB:000280</td>
</tr>
<tr>
<td>+000046</td>
<td>JFCBVLSQ:0000 JFCBIND1:00 JFCBIND2:40</td>
</tr>
<tr>
<td>+00005B</td>
<td>JFCBUFNO:00 JFCDSR1:00 JFCDSR2:00</td>
</tr>
<tr>
<td>+000064</td>
<td>JFCRECM:00 JFCBLKSI:0000 JFCRECL:0000 JFCNCP:00</td>
</tr>
<tr>
<td>+000075</td>
<td>JFCBNVOL:01 JFCBNVOLS:SLBD1C</td>
</tr>
<tr>
<td>+000094</td>
<td>JFCBEOTL:00 JFCBEAD:000000 JFCFLGS1:00</td>
</tr>
<tr>
<td>+0000AE</td>
<td>JFCBVLC1:01</td>
</tr>
</tbody>
</table>

**Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 16 of 22)**
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 17 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 18 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 19 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 20 of 22)
Figure 19. Example formatted C/C++ output from LEDATA Verbexit (Part 21 of 22)
### C/C++-specific sections of the LEDATA output

For the LEDATA output:

1. **CGEN**
   - This section formats the C/C++-specific portion of the Language Environment common anchor area (CAA).

2. **CGENE**
   - This section formats the extension to the C/C++-specific portion of the Language Environment common anchor area (CAA).

3. **CEDB**
   - Dummy FCB encountered at location 20C0E148

```
<table>
<thead>
<tr>
<th>AMRC: 20C0E770</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE:002170B</td>
</tr>
<tr>
<td>RBA:00000000</td>
</tr>
<tr>
<td>LASTOP:00000032</td>
</tr>
<tr>
<td>+000000 FILL_LEN:00000000</td>
</tr>
<tr>
<td>MSG_LEN:00000000</td>
</tr>
<tr>
<td>+000014 STR1:..........................</td>
</tr>
<tr>
<td>+000050 STR1_CONT:..........................</td>
</tr>
<tr>
<td>+0000BC PARMR0:00000000</td>
</tr>
<tr>
<td>PARMR1:00000000</td>
</tr>
<tr>
<td>+00009C STR2:..........................</td>
</tr>
<tr>
<td>+0000DC RPLFDW0:00000000</td>
</tr>
<tr>
<td>XBA:00000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>+0000EB AMRC_NOSEEK_TOSEEK:00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AMRC2: 20C0E87B</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 __ERROR2:00000000</td>
</tr>
<tr>
<td>__FILEPTR:00000000</td>
</tr>
</tbody>
</table>

File name: CHARUM.A.B

8. **MFCB: 2135A470**
   - FIRSTNEBULA:2135A4E4  |
   | RFECNT:00000001  |
   | NEXTMFCB:2135A970  |
   |+000010 WRITEFCB:2135A268  |
   | FLAG1:0001  |
   | DEPTH:0000  |
   | NAME:2135A5F0  |
   |+0000C NAMELENGTH:0000000A  |
   | NAMEBUFSIZE:00000044  |
   | MEMBER:........ |
   |+00002C NEXTFCB:00000000  |
   | PREVFCB:00000000  |
   |+000034 PARENTFCB:2135A268  |
   | CHILDFCB:00000000  |
   |+00003C PREVMFCB:00000000  |
   | PARENTMFCB:2135A470  |
   |+000044 CHILDMFCB:00000000  |
   | HPERKEY:00000000  |
   | 00000000  |
   |+000050 CURHSPBYTES:00000000  |
   | LASTBYTE:0000  |
   | CREATELEVEL:0000 |
   |+000064 LASTDS:00000000  |
   | MAXHSPBYTES:00000000  |
   |+00006C LASTBLKOFFSET:00000000  |
   | MFCB_CPCB:20C0DEA8 |

File name: CHARUM.B.C

<table>
<thead>
<tr>
<th>MFCB: 2135A970</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 FIRSTNEBULA:00000000</td>
</tr>
<tr>
<td>RFECNT:00000001</td>
</tr>
<tr>
<td>NEXTMFCB:00000000</td>
</tr>
<tr>
<td>+000010 WRITEFCB:2135A660</td>
</tr>
<tr>
<td>FLAG1:0001</td>
</tr>
<tr>
<td>DEPTH:0000</td>
</tr>
<tr>
<td>NAME:2135A9F0</td>
</tr>
<tr>
<td>+00001C NAMELENGTH:00000000</td>
</tr>
<tr>
<td>NAMEBUFSIZE:00000044</td>
</tr>
<tr>
<td>MEMBER:........</td>
</tr>
<tr>
<td>+00002C NEXTFCB:00000000</td>
</tr>
<tr>
<td>PREVFCB:00000000</td>
</tr>
<tr>
<td>+000034 PARENTFCB:2135A660</td>
</tr>
<tr>
<td>CHILDFCB:00000000</td>
</tr>
<tr>
<td>+00003C PREVMFCB:00000000</td>
</tr>
<tr>
<td>PARENTMFCB:2135A470</td>
</tr>
<tr>
<td>+000044 CHILDMFCB:00000000</td>
</tr>
<tr>
<td>HPERKEY:80007400</td>
</tr>
<tr>
<td>0011F4C</td>
</tr>
<tr>
<td>+000050 CURHSPBYTES:00000000</td>
</tr>
<tr>
<td>LASTBYTE:0000</td>
</tr>
<tr>
<td>CREATELEVEL:0000</td>
</tr>
<tr>
<td>+000058 FLAG2:40000000</td>
</tr>
<tr>
<td>LASTNEBULA:00000000</td>
</tr>
<tr>
<td>LASTNEBINDEX:0000</td>
</tr>
<tr>
<td>+000064 LASTDS:00000000</td>
</tr>
<tr>
<td>MAXHSPBYTES:00000000</td>
</tr>
<tr>
<td>+00006C LASTBLKOFFSET:00000000</td>
</tr>
<tr>
<td>MFCB_CPCB:20C0DEA8</td>
</tr>
</tbody>
</table>

Exiting CRTL I/O Control Blocks
Exiting Language Environment Data

---

**Figure 19.** Example formatted C/C++ output from LEDATA Verbexit (Part 22 of 22)
This section formats the C/C++-specific portion of the Language Environment enclave data block (EDB).

[4] CTHD

This section formats the C/C++ thread-level control block (CTHD).

[5] CPCB

This section formats the C/C++-specific portion of the Language Environment process control block (PCB).

[6] CIO

This section formats the C/C++ IO control block (CIO).

[7] File Control Blocks

This section formats the C/C++ file control block (FCB). The FCB and its related control blocks represent the information needed by each open stream.

Related Control Blocks

FFIL  This section formats the header of the C/C++ file control block (FCB).

FSCE  The file specific category extension control block. The FSCE represents the specific type of IO being performed. The following is a list of FSCEs that may be formatted.

HFSF — UNIX file system file
HSPF — Hiper-Space file
INTC — Intercept file
MEMF — Memory file
OSNS — OS no seek
OSFS — OS fixed text
OSVF — OS variable text
OSUT — OS undefined format text
TDQF — CICS Transient Data Queue file
TERM — Terminal file
VSAM — VSAM file

Other FSCEs will be displayed using a generic overlay.

OSIO  The OS IO interface control block.

OSIOE The OS IO extended interface control block.

DCB  The data control block. For more information about the DCB, see [z/OS DFSMS Macro Instructions for Data Sets]

DCBE The data control block extension. For more information about the DCBE, see [z/OS DFSMS Macro Instructions for Data Sets]
JFCB  The job file control block (JFCB). For more information about the JFCB, see z/OS MVS Data Areas, Vol 3 (IVT-RCWK).

JFCBX  The job file control block extension (JFCBX).

MBUF  The message buffer control block (MBUF).

[8] Memory File Control Blocks

This section formats the C/C++ memory file control block (MFCB).

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 ALL parameter is specified and COBOL is active in the dump. Figure 20 on page 148 illustrates the COBOL-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) run-time option. “COBOL-specific sections of the LEDATA Output” on page 149 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 COBOL output from LEDATA Verbexit (Part 1 of 2)
Figure 20. Example formatted COBOL output from LEDATA Verbexit (Part 2 of 2)

COBOL-specific sections of the LEDATA Output

For the LEDATA output:

[1] RUNCOM
This section formats the COBOL enclave-level control block (RUNCOM).

[2] THDCOM
This section formats the COBOL process-level control block (THDCOM).

[3] COBCOM
This section formats the COBOL region-level control block (COBCOM).

[4] CLLE
This section formats the COBOL loaded program control blocks (CLLE).

[5] TGT
This section formats the COBOL TGT control blocks.

**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".

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 verb exit.

*cbname*

The name of the control block to be formatted. The control blocks that can be individually formatted are listed in Table 8 on page 151. 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 in order to uniquely define the Language Environment control block names to IPCS.

For an example of the display which is the result of the command, see Figure 21 on page 151:

```
CBF 15890 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 8. Language Environment Control blocks that can be individually formatted

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEADHP</td>
<td>Additional Heap Control Block</td>
</tr>
<tr>
<td>CEECAA</td>
<td>Common Anchor Area</td>
</tr>
<tr>
<td>CEECIB</td>
<td>Condition Information Block</td>
</tr>
<tr>
<td>CEECIBH</td>
<td>Condition Information Block Header</td>
</tr>
<tr>
<td>CEECMXB</td>
<td>Message Services Block</td>
</tr>
<tr>
<td>CEEDSA</td>
<td>Dynamic Storage Area</td>
</tr>
<tr>
<td>CEEDLLF</td>
<td>DLL Failure Control Block</td>
</tr>
<tr>
<td>CEEDSATR</td>
<td>XPLINK Transition Area</td>
</tr>
<tr>
<td>CEEDSAX</td>
<td>Dynamic Storage Area (XPLINK style)</td>
</tr>
<tr>
<td>CEEEDB</td>
<td>Enclave Data Block</td>
</tr>
<tr>
<td>CEEENSM</td>
<td>Enclave Level Storage Management</td>
</tr>
</tbody>
</table>

Figure 21. The CAA formatted by the CBFORMAT IPCS command

For more information on using the IPCS CBF command refer to the “CBFORMAT subcommand” section in z/OS MVS IPCS Commands, SA22-7594.
Table 8. 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>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>CEEPMBB</td>
<td>Process Control Block</td>
</tr>
<tr>
<td>CEEPMBB</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. Under abnormal termination, the following dump contents are generated:

- TERMTHDACT(QUIET) generates a Language Environment dump containing the trace table only
TERMTHDACT(MSG) generates a Language Environment dump containing the trace table only
TERMTHDACT(TRACE) generates a Language Environment dump containing the trace table and the traceback
TERMTHDACT(DUMP) generates a Language Environment dump containing thread/enclave/process storage and control blocks (the trace table is included as an enclave control block)
TERMTHDACT(UAONLY) generates a system dump of the user address space
TERMTHDACT(UATRACE) generates a Language Environment dump that contains traceback information, and a system dump of the user address space
TERMTHDACT(UADUMP) generates a 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
TERMTHDACT(UAIMM) generates a system dump of the user address space of the original abend or program interrupt that occurred prior to the Language Environment condition manager processing the condition.

Note: Under CICS, UAIMM yields UAONLY behavior. Under non-CICS, TRAP(ON,NOSPIE) must be in effect. When TRAP(ON,SPIE) is in effect, UAIMM yields UAONLY behavior. For software raised conditions or signals, UAIMM behaves the same as UAONLY.

Under normal termination, the following dump contents are generated:
• 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
```

<table>
<thead>
<tr>
<th>path</th>
<th>The path determined from the above algorithm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fname</td>
<td>The name specified in the FNAME parameter on the call to CEE3DMP (default is CEEDUMP).</td>
</tr>
<tr>
<td>Date</td>
<td>The date the dump is taken, appearing in the format YYYYMMDD (such as 20040918 for September 18, 2004).</td>
</tr>
<tr>
<td>Time</td>
<td>The time the dump is taken, appearing in the format HHMMSS (such as 175501 for 05:55:01 p.m.).</td>
</tr>
<tr>
<td>Pid</td>
<td>The process ID the application is running in when the dump is taken.</td>
</tr>
</tbody>
</table>

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:

```
Time of Day | Thread ID | Member ID and Flags | Member entry type | Mbr-specific info up to a maximum of 104 bytes
Char (8)     | Char (8)  | Char (4)            | Char (4)          | Char (104)
```

Figure 22. Format of the trace table entry

Following is a definition of each field:

**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:
  - ID 01 Name CEL
  - ID 03 Name C/C++
  - ID 05 Name COBOL
  - ID 07 Name Fortran
<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: The following C/C++ records may be generated.

Table 9. LE=1 entry records

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000001</td>
<td>Base C Library function Entry</td>
</tr>
<tr>
<td>03</td>
<td>00000002</td>
<td>Base C Library function Exit</td>
</tr>
<tr>
<td>03</td>
<td>00000003</td>
<td>Posix C Library function Entry</td>
</tr>
<tr>
<td>03</td>
<td>00000004</td>
<td>Posix C Library function Exit</td>
</tr>
<tr>
<td>03</td>
<td>00000005</td>
<td>XPLINK Base or Posix C Library function Entry</td>
</tr>
<tr>
<td>03</td>
<td>00000006</td>
<td>XPLINK Base or Posix C Library function Exit</td>
</tr>
</tbody>
</table>

For a detailed description of these records, see "C/C++ contents of the Language Environment trace tables" on page 207.
When \( LE=2 \) is specified: The following Language Environment records may be generated.

**Table 10. \( LE=2 \) 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>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)</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>
</tbody>
</table>
Table 10. 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>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 erpm</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>EOU</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>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>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 SMCDESTROY</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>
</tbody>
</table>
## Table 10. 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>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>
<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>
</tbody>
</table>
Table 10. 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>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>

The format for the Mutex – Condition Variable – Latch entries in the trace table is:

Table 11. 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>

Where each field represents:

**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++
- **D**: DCE
- **S**: Sockets

**Blank**

Blank character

**Event**

Two character EBCDIC representation of the event. See Table 10 on page [156](#).

**Object address**

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. Currently this is only supported by C/C++.

Table 12. 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>

For a detailed description of these records, see "C/C++ contents of the Language Environment trace tables" on page 207.

When LE=20 is specified: The following C/C++ records might be generated.

Table 13. 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>

For a detailed description of these records, see "C/C++ contents of the Language Environment trace tables" on page 207.

Sample dump for the trace table entry

The following is 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
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.
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.

typedef struct __amrctype {
    union {
        long int __error;
        struct {
            unsigned short __syscode,
            __rc;
        } __abend;
        struct {
            unsigned char __fdbk_fill,
            __rc,
            __ftncd,
            __fdbk;
        } __feedback;
        struct {
            unsigned short __svc99_info,
            __svc99_error;
        } __alloc;
        union {
            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
        } __code;
        unsigned long __RBA;
        unsigned int __last_op;
    } __amrc_type;
}

Figure 24. __amrc structure

Figure 25 on page 169 shows the __amrc2 structure as it appears in stdio.h.
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.

__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.

__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.

__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.

__alloc
A structure that contains errors during fopen or freopen calls when defining files to the system using SVC 99.

__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.

__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 14 on page 170.

__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.

__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>Macro</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------</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 14 lists __last_op values you could receive and where to look for further information.

Table 14. __last_op values and diagnosis information

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__IO_INIT</td>
<td>Will never be seen by SIGIOERR exit value given at initialization.</td>
</tr>
<tr>
<td>__BSAM_OPEN</td>
<td>Sets __error with return code from OS OPEN macro.</td>
</tr>
<tr>
<td>__BSAM_CLOSE</td>
<td>Sets __error with return code from OS CLOSE macro.</td>
</tr>
<tr>
<td>__BSAM_READ</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 66) filled in).</td>
</tr>
<tr>
<td>__BSAM_NOTE</td>
<td>NOTE returned 0 unexpectedly, no return code.</td>
</tr>
<tr>
<td>__BSAM_POINT</td>
<td>This will not appear as an error lastop.</td>
</tr>
<tr>
<td>__BSAM_WRITE</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 65) filled in).</td>
</tr>
<tr>
<td>__BSAM_CLOSE_T</td>
<td>Sets __error with return code from OS CLOSE TYPE=T.</td>
</tr>
<tr>
<td>__BSAM_BDL</td>
<td>Sets __error with return code from OS BDL macro.</td>
</tr>
<tr>
<td>__BSAM_STOW</td>
<td>Sets __error with return code from OS STOW macro.</td>
</tr>
<tr>
<td>__TGET_READ</td>
<td>Sets __error with return code from TSO TGET macro.</td>
</tr>
<tr>
<td>__TPUT_WRITE</td>
<td>Sets __error with return code from TSO TPUT macro.</td>
</tr>
</tbody>
</table>
Table 14. __last_op values and diagnosis information  (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__I0_DEVTYPE</td>
<td>Sets __error with return code from I/O DEVTYPE macro.</td>
</tr>
<tr>
<td>__I0_RDJFCB</td>
<td>Sets __error with return code from I/O RDJFCB macro.</td>
</tr>
<tr>
<td>__I0_TRKCALC</td>
<td>Sets __error with return code from I/O TRKCALC macro.</td>
</tr>
<tr>
<td>__I0_OBTAIN</td>
<td>Sets __error with return code from I/O CAMLST OBTAIN.</td>
</tr>
<tr>
<td>__I0_LOCATE</td>
<td>Sets __error with return code from I/O CAMLST LOCATE.</td>
</tr>
<tr>
<td>__I0_CATALOG</td>
<td>Sets __error with return code from I/O CAMLST CAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>__I0_UNCATALOG</td>
<td>Sets __error with return code from I/O CAMLST UNCAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>__I0_RENAME</td>
<td>Sets __error with return code from I/O CAMLST RENAME.</td>
</tr>
<tr>
<td>__SVC99_ALLOC</td>
<td>Sets __alloc structure with info and error codes from SVC 99 allocation.</td>
</tr>
<tr>
<td>__SVC99_ALLOC_NEW</td>
<td>Sets __alloc structure with info and error codes from SVC 99 allocation of NEW file.</td>
</tr>
<tr>
<td>__SVC99_UNALLOC</td>
<td>Sets __unalloc structure with info and error codes from SVC 99 unallocation.</td>
</tr>
<tr>
<td>__C_TRUNCATE</td>
<td>Set when C or C++ truncates output data. Usually this is data written to a text file with no newline such that the record fills up to capacity and subsequent characters cannot be written. For a record I/O file this refers to an fwrite() writing more data than the record can hold. Truncation is always rightmost data. There is no return code.</td>
</tr>
<tr>
<td>__C_FCBCHECK</td>
<td>Set when C or C++ FCB is 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_UUNEVEN</td>
<td>This occurs when an SI is written before the last double byte character is completed, thereby forcing C or C++ to fill in the last byte of the DBCS string with a padding byte X'FE'. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_CANNOT_EXTEND</td>
<td>This occurs when an attempt is made to extend a file that allows writing, but cannot be extended. Typically this is a member of a partitioned data set being opened for update.</td>
</tr>
<tr>
<td>__VSAM_OPEN_FAIL</td>
<td>Set when a low level VSAM OPEN fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_RRDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__VSAM_OPEN_KSDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is 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>__CICS_WRITEQ_TD</td>
<td>Sets __error with error code from EXEC CICS WRITEQ TD.</td>
</tr>
<tr>
<td>__LFS_OPEN</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in <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>
<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 <a href="#">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</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 <a href="#">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
<tr>
<td>__LFS_FSTAT</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in <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). 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 26 is an example of a routine using perror().

```c
#include <stdio.h>

int main()
{
    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.

Figure 27 on page 175 is an example of a routine using __errno2() and Figure 28 on page 175 shows the sample output from that routine.
Figure 29 is an example of a routine using the environment variable \_EDC\_ADD\_ERRNO2 and Figure 30 shows the sample output from that routine.

```
#pragma runopts(posix(on))
#define _EXT
#include <stdio.h>
#include <errno.h>

int main(void) {
    FILE *f;
    f = fopen("testfile.dat", "r")
    if (f==NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    }
    return 0;
}
```

Figure 27. Example of a routine using __errno2()

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

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

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(void) {
    FILE *fp;
    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

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

Figure 30. Sample output of a routine using _EDC\_ADD\_ERRNO2

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.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 31 on page 176 is an example of a routine using __err2ad() in combination with __errno2() and Figure 32 on page 176 shows the sample output from that routine.
#pragma runopts(posix(on))
#define _EXT
#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\n", __errno2());
    }
    /* reset errno2 to zero */
    *__err2ad() = 0x0;
    printf("__errno2 = %08x\n", __errno2());
    f = fopen(*testfile.dat", "r");
    if (fp == NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    }
    return 0;
}

Figure 31. Example of a routine using __err2ad() in combination with __errno2()

fopen() failed: EDC5129I No such file or directory.
   __errno2 = 05620062
   __errno2 = 00000000
fopen() failed: EDC5129I No such file or directory.
   __errno2 = 05620062

Figure 32. Sample output of routine using __err2ad() in combination with __errno2()

For more information about _EDC_ADD_ERRNO2, see z/OS XL C/C++ Programming Guide.

For more information about __errno2() and __err2ad(), see z/OS XL C/C++ 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.

   Example:
   
   aal  85-0:85  Class = automatic, Offset = 164(r13), Length = 40

   4. Add this base address to the offset of the variable.

   When you are done, the contents of the variable can be read in the DSA Frame section corresponding to the function the variable is contained in.

Locating the Writable Static Area (WSA)

The Writable Static Area (WSA) address is the base address of the writable static area which is available for all C and C++ programs except C programs compiled with the NORENT compiler option. If you have C code compiled with the RENT option or C++ code (hereafter called RENT code) you must determine the base address of the WSA if you want to calculate the address of a static or external variable. Use the following table to determine where to find the WSA base address:

<table>
<thead>
<tr>
<th>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 185.

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 177. 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.

   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.
   Example: X'02D66E40' + X'58' = X'2D66E98'

3. Find the offset of the static variable in the partial storage offset compiler listing.
   Example:
   sa0 66-0:66 Class = static, Location = WSA + @STATIC + 96, Length = 4

   The offset is 96 (X'60').

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).
   Example: 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 on page 180 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 177. 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.
   **Example:**
   In this example, the offset for DFHEIPTR is X'28'.

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>LENGTH</th>
<th>FILE ID</th>
<th>INPUT NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>00001</td>
<td>DFHC0011</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>00001</td>
<td>DFHC0010</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>00001</td>
<td>DFHDUMMY</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>00001</td>
<td>DFHB0025</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>00001</td>
<td>DFHB0024</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>00001</td>
<td>DFHB0023</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>00001</td>
<td>DFHB0022</td>
</tr>
<tr>
<td>1C</td>
<td>2</td>
<td>00001</td>
<td>DFHB0021</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>00001</td>
<td>DFHB0020</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>00001</td>
<td>DFHEIB0</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>00001</td>
<td>DFHEIPTR</td>
</tr>
<tr>
<td>2C</td>
<td>4</td>
<td>00001</td>
<td>DFHCP011</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>00001</td>
<td>DFHCP010</td>
</tr>
<tr>
<td>34</td>
<td>4</td>
<td>00001</td>
<td>DFHB0025</td>
</tr>
<tr>
<td>38</td>
<td>4</td>
<td>00001</td>
<td>DFHB0024</td>
</tr>
<tr>
<td>3C</td>
<td>4</td>
<td>00001</td>
<td>DFHB0023</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>00001</td>
<td>DFHB0022</td>
</tr>
<tr>
<td>44</td>
<td>4</td>
<td>00001</td>
<td>DFHB0021</td>
</tr>
<tr>
<td>48</td>
<td>4</td>
<td>00001</td>
<td>DFHB0020</td>
</tr>
<tr>
<td>4C</td>
<td>4</td>
<td>00001</td>
<td>DFHICB</td>
</tr>
<tr>
<td>50</td>
<td>4</td>
<td>00001</td>
<td>DFHEID0</td>
</tr>
<tr>
<td>54</td>
<td>4</td>
<td>00001</td>
<td>DFHLDVER</td>
</tr>
<tr>
<td>58</td>
<td>420</td>
<td>00001</td>
<td>@STATIC</td>
</tr>
</tbody>
</table>

2. Add the offset of the external variable to the address of writable static.
   **Example:** X'02D66E40' + X'28' = X'02D66E68'

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 178. 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.
   
   **Example:**
   
   sa0 66-0:66 Class = static,  Location = STATSTOR +96,  Length = 4
   
   The offset is 96 (X'60').

2. Add the offset to the base address of static variables.
   
   **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 external NORENT variables

Before you begin: You need to find the address of the external variable CSECT. To find this information, see "Steps for finding the static storage area" on page 178. For this procedure’s example, the address of the external variable CSECT is X'02D66E40'.

The address of the external variable CSECT is the address of the value of the external variable in the Language Environment dump.

Steps for finding the C/370 parameter list

Perform the following steps to locate a parameter in the Language Environment dump:

1. Indentify 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 on page 182](#) shows an example code for the parameter variable.

   **Example:**

---
Parameters \( ppx \) and \( pp0 \) correspond to copies of \( a1 \) and \( a2 \) in the stack frame belonging to \( \text{func0}() \).

2. Use the address of the start of the parameter list to find the register and offset in the partial storage offset listing.

   **Example:**
   
   \[
   \begin{array}{ll}
   ppx & 62-0:62 \\
   \text{Class} & = \text{parameter}, \quad \text{Location} = 4(r1), \quad \text{Length} = 4
   \end{array}
   \]
   
   The offset is 4 (\( \text{X}'4' \)) from register 1.

3. Determine the value of \( \text{GPR1} \) 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 \( \text{extern C} \) attributes, see "Steps for finding the C/370 parameter list" on page 181.

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.

   **Example:**

   ```
   \begin{verbatim}
   func0() {
   ... 
   func1(a1,a2); 
   ... 
   }
   func1(int ppx, int pp0) {
   ... 
   }
   \end{verbatim}
   ```

   **Figure 38. Example code for parameter variable**

   Parameters \( ppx \) and \( pp0 \) correspond to copies of \( a1 \) and \( a2 \) in the stack frame belonging to \( \text{func1}() \).
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.

   Example:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>62-0:62</td>
<td>Class = parameter,</td>
<td>Location = 188(r13),</td>
<td>Length = 4</td>
</tr>
<tr>
<td>62-0:62</td>
<td>Class = parameter,</td>
<td>Location = 192(r13),</td>
<td>Length = 4</td>
</tr>
</tbody>
</table>

   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** shows an example of a static aggregate.

```c
static struct {
    short int ss01;
    char ss02[56];
    int sz0[6];
    int ss03;
} ss0;
```

**Figure 40. Example code for structure variable**

**Figure 41 on page 184** shows an example aggregate map.
Assume the structure has been compiled as RENT. To find the value of variable sz0[0]:
1. Find the address of the writable static. For this example the address of writable static is X'02D66E40'.
2. Find the offset of @STATIC in the Writable Static Map. In this example, the offset is X'58'. Add this offset to the address of writable static. The result is X'2D66E98' (X'02D66E40' + X'58' ). Figure 42 shows the Writable Static Map produced by the prelinker.

---

Figure 41. Example of aggregate map

Assume the structure has been compiled as RENT. To find the value of variable sz0[0]:
1. Find the address of the writable static. For this example the address of writable static is X'02D66E40'.
2. Find the offset of @STATIC in the Writable Static Map. In this example, the offset is X'58'. Add this offset to the address of writable static. The result is X'2D66E98' (X'02D66E40' + X'58' ). Figure 42 shows the Writable Static Map produced by the prelinker.

---

Figure 42. Writable static map produced by prelinker
3. Find the offset of the static variable in the storage offset listing. The offset is 96 (X'60'). 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 184. 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 either the CEE3DMP callable service or the cdump(), csnap(), and ctrace() C/C++ functions to generate a Language Environment dump of C/C++ routines. These C/C++ functions call CEE3DMP with specific options.

**cdump()**

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 this message:

```
Snap was successful; snap ID = mnn
```
Where \textit{nnn} corresponds to the SNAP identifier described above. An unsuccessful SNAP does not result in an incrementation of the identifier.

Because \texttt{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 \texttt{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 \_\texttt{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 \texttt{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.

\textbf{csnap()}

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

To use these functions, you must add \texttt{#include <ctest.h>} to your C/C++ code. The dump is directed to output \textit{dumpname}, which is specified in a \texttt{//CEEDUMP DD} statement in MVS/JCL.

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

See the \textit{z/OS XL C/C++ Run-Time Library Reference} for more details about the syntax of these functions.

\textbf{ctrace()}

The \texttt{ctrace()} function produces a traceback and includes the offset addresses from which the calls were made. \texttt{ctrace()} is equivalent to calling CEE3DMP with the option string: \texttt{TRACEBACK NOFILES NOBLOCKS NOVARIABLES NOSTORAGE STACKFRAME(ALL) NOCONDITION NOENTRY}.

\textbf{Sample C routine that calls \texttt{cdump()}}

Figure 43 on page 187 shows a sample C routine that uses the \texttt{cdump} function to generate a dump.

Figure 48 on page 191 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);
    }
}"

---

Figure 43. Example C routine using cdump() to generate a dump (Part 1 of 2)
Figure 43. Example C routine using cdump() to generate a dump (Part 2 of 2)

```
#include <ctest.h>
#pragma linkage(func1, fetchable)
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 189](#) 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*  

*Figure 47 on page 190 shows the template file stack.c*
Sample Language Environment dump with C/C++-specific information

This sample dump was produced by compiling the routine in Figure 43 on page 187 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. For more information about C/C++-specific information contained in a dump, see "Finding C/C++ information in a Language Environment dump" on page 196.
Figure 48. Example dump from sample C routine (Part 1 of 6)
Control Blocks for Active Routines:

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>20FCB6B</td>
<td>000000</td>
<td>000000</td>
<td>DSA for func: C-C++</td>
</tr>
<tr>
<td>20FCB68</td>
<td>000000</td>
<td>000000</td>
<td>+000000</td>
</tr>
<tr>
<td>20FCB68</td>
<td>000010</td>
<td>000010</td>
<td>+000010</td>
</tr>
<tr>
<td>20FCB68</td>
<td>000024</td>
<td>000024</td>
<td>+000024</td>
</tr>
<tr>
<td>20FCB68</td>
<td>000038</td>
<td>000038</td>
<td>+000038</td>
</tr>
<tr>
<td>20FCB68</td>
<td>000044</td>
<td>000044</td>
<td>+000044</td>
</tr>
<tr>
<td>20FCB68</td>
<td>000078</td>
<td>000078</td>
<td>+000078</td>
</tr>
</tbody>
</table>

Storage for Active Routines:

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>20FCB4B</td>
<td>000000</td>
<td>000000</td>
<td>DSA frame: C-C++</td>
</tr>
<tr>
<td>20FCB4B</td>
<td>000020</td>
<td>000020</td>
<td>+000020</td>
</tr>
<tr>
<td>20FCB4B</td>
<td>000040</td>
<td>000040</td>
<td>+000040</td>
</tr>
<tr>
<td>20FCB4B</td>
<td>000060</td>
<td>000060</td>
<td>+000060</td>
</tr>
<tr>
<td>20FCB4B</td>
<td>000080</td>
<td>000080</td>
<td>+000080</td>
</tr>
</tbody>
</table>

Control Blocks Associated with the Thread:

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>20FCB6B</td>
<td>000000</td>
<td>000000</td>
<td>CAAA: 20FEB058</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000020</td>
<td>000020</td>
<td>+000020</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000040</td>
<td>000040</td>
<td>+000040</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000060</td>
<td>000060</td>
<td>+000060</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000080</td>
<td>000080</td>
<td>+000080</td>
</tr>
</tbody>
</table>

CAA C-C++ information:

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>20FCB6B</td>
<td>000000</td>
<td>000000</td>
<td>C-C++ Specific CTHD: 20FEB058</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000040</td>
<td>000040</td>
<td>C-C++ Specific CDB: 20FEB058</td>
</tr>
</tbody>
</table>

C-C++ Specific Thread block: 20FEB058

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>20FCB6B</td>
<td>000000</td>
<td>000000</td>
<td>+000000</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000010</td>
<td>000010</td>
<td>+000010</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000020</td>
<td>000020</td>
<td>+000020</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000030</td>
<td>000030</td>
<td>+000030</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000040</td>
<td>000040</td>
<td>+000040</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000050</td>
<td>000050</td>
<td>+000050</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000060</td>
<td>000060</td>
<td>+000060</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000070</td>
<td>000070</td>
<td>+000070</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000080</td>
<td>000080</td>
<td>+000080</td>
</tr>
</tbody>
</table>

C-C++ Specific EDB block: 20FEB08 |

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>20FCB6B</td>
<td>000000</td>
<td>000000</td>
<td>+000000</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000010</td>
<td>000010</td>
<td>+000010</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000020</td>
<td>000020</td>
<td>+000020</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000030</td>
<td>000030</td>
<td>+000030</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000040</td>
<td>000040</td>
<td>+000040</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000050</td>
<td>000050</td>
<td>+000050</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000060</td>
<td>000060</td>
<td>+000060</td>
</tr>
<tr>
<td>20FCB6B</td>
<td>000070</td>
<td>000070</td>
<td>+000070</td>
</tr>
</tbody>
</table>

Figure 48. Example dump from sample C routine (Part 2 of 6)
Figure 48. Example dump from sample C routine (Part 3 of 6)
Figure 48. Example dump from sample C routine (Part 4 of 6)
fdata FOR FILE: 'HEALY.MYFILE.DATA'
__recfmF:1........ 0
__recfmV:1........ 1
__recfmU:1........ 0
__recfmS:1........ 0
__recfmBlk:1...... 1
__recfmASA:1...... 0
__recfmM:1........ 0
__recfmPO:1....... 0
__dsorgPDSmem:1... 0
__dsorgPDSdir:1... 0
__dsorgPS:1....... 1
__dsorgConcat:1... 0
__dsorgMem:1...... 0
__dsorgHiper:1.... 0
__dsorgTemp:1..... 0
__dsorgVSAM:1..... 0
__dsorgHFS:1...... 0
__openmode:2...... 0
__modeflag:4...... 2
__dsorgPDSE:1..... 0
__reserve2:4...... 0
__device......... 0
__blksize........ 6144
__maxreclen....... 1024
__access_method... 1(1)
__noseek_to_seek.. 0(0)
__dsname......... HEALY.MYFILE.DATA
FILE pointer........ 20FF4024
dname.............. SYS00001
Buffer at current file position: 20FF42E8
+000000 20FF42E8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | record 1...record 2... record 3
+000040 20FF42E8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | same as above
Saved Buffer........ NULL
Write Data Control Block: 00015020
+000000 00015020 20FF42E8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ...y............Vs.......
+000040 00015020 20FF42E8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | same as above
read/update DCEB..... NULL
Write Data Control Block Extension: 20FF42A8
+000000 20FF42A8 00015020 20FF42A8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | ...f...&........d......S.........|
+000020 20FF42A8 00015020 20FF42A8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | same as above
read/update DCBE..... NULL
Job File Control Block: 000157EB
+000000 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB | HEALY.MYFILE.DATA
+000020 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB | HEALY.MYFILE.DATA
+000040 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB 000157EB | HEALY.MYFILE.DATA

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.

[1] Storage for Active Routines

The Storage for Active Routines section of the dump 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'`.

[2] Control Blocks Associated with the Active Thread
In the Control Blocks Associated with the Thread section of the dump, the following information appears:

- Fields from the CAA
- Fields specific from the CTHD and CEDB
- Signal information

[2A] C/C++ CAA Fields

The CAA 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 [198].

**Open FCB chain**
- A pointer to the start of a linked list of open file control blocks (FCBs). For more information about FCBs, see [198].

[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

The 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.

Chapter 4. Debugging C/C++ routines 197
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**

The `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.

```c
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](#).

[7] **File Control Block Information**

This section of the dump includes the file control block (FCB) information for each C/C++ file. The FCB contains file status and attributes for files open during C/C++ active routines. You can use this information to find the data set or file name.

The FCB is a handle that points to the following file information, which is displayed when applicable, for the file:

- Access method control block (ACB) address
- Data control block (DCB) address
- Data control block extension (DCBE) address
- Job file control block (JFCB) address
- RPL address
- Current buffer address
- Saved buffer address
- ddname

Not all FCB fields are always filled in. For example, RPLs are used only for VSAM data sets. The `ddname` field contains blanks if it is not used.

The save block buffer represents auxiliary buffers that are used to save the contents of the main buffers. Such saving occurs only when a reposition is performed and there is new data; for example, an incomplete text record or an incomplete fixed-block standard (FBS) block in the buffers that cannot be flushed out of the system.

Because the main buffers represent the current position in the file, while the save buffers merely indicate a save has occurred, check the save buffers only if data appears to be missing from the external device and is not found in the main buffers. Also, do not infer that the presence of save buffers means that data present there belongs at the end of the file. (The buffers remain, even when the data is eventually written.)

For information about the job file control block, see [z/OS MVS Data Areas, Vol 3 (IVT-RCWK)](#).

**Memory File Control Block**

This section of the dump holds the memory file control block information for each memory file the routine uses. A sample memory file control block is shown in...
Figure 49. Memory file name

<table>
<thead>
<tr>
<th>Memory file name</th>
<th>The name assigned to this memory file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>First memory data space</td>
<td>A dump of the first 1K maximum of actual user data associated with this memory file.</td>
</tr>
</tbody>
</table>

[8] Information for __amrc

__amrc is a structure defined in the stdio.h header file to assist in determining errors resulting from I/O operations. The contents of __amrc can be checked for system information, such as the return code for VSAM. Certain fields of the __amrc structure can provide useful information about what occurred previously in your routine.

For more information about __amrc, see "Debugging C/C++ programs" on page 167 and to z/OS XL C/C++ Programming Guide.

[9] Errno Information

The Errno information shows the thread id of the thread that generated the dump and the settings of the errno and errnojr variables for that thread.

Both the errno and the errnojr variables contain the return code of the last failing z/OS UNIX system service call. These variables provide z/OS UNIX application programs access to diagnostic information returned from an underlying z/OS UNIX callable service. For more information on these return and reason codes, see z/OS UNIX System Services Messages and Codes.

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 twelve floating-point registers. A sample output is shown.
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.

**Figure 50. Registers on entry to CEE3DMP**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM</td>
<td>0100</td>
</tr>
<tr>
<td>GPR0</td>
<td>183F88E8</td>
</tr>
<tr>
<td>GPR1</td>
<td>00023D38</td>
</tr>
<tr>
<td>GPR2</td>
<td>00023E98</td>
</tr>
<tr>
<td>GPR3</td>
<td>1840E792</td>
</tr>
<tr>
<td>GPR4</td>
<td>0002309B</td>
</tr>
<tr>
<td>GPR5</td>
<td>183F8CD0</td>
</tr>
<tr>
<td>GPR6</td>
<td>00023D48</td>
</tr>
<tr>
<td>GPR7</td>
<td>0002297F</td>
</tr>
<tr>
<td>GPR8</td>
<td>17F4553D</td>
</tr>
<tr>
<td>GPR9</td>
<td>183F6670</td>
</tr>
<tr>
<td>GPR10</td>
<td>17F4353F</td>
</tr>
<tr>
<td>GPR11</td>
<td>17FA0550</td>
</tr>
<tr>
<td>GPR12</td>
<td>00015920</td>
</tr>
<tr>
<td>GPR13</td>
<td>00023CA0</td>
</tr>
<tr>
<td>GPR14</td>
<td>800180E2</td>
</tr>
<tr>
<td>GPR15</td>
<td>97F57FE8</td>
</tr>
<tr>
<td>FPC</td>
<td>40084000</td>
</tr>
<tr>
<td>FPR0</td>
<td>40260000</td>
</tr>
<tr>
<td>FPR1</td>
<td>41086A00</td>
</tr>
<tr>
<td>FPR2</td>
<td>3FBCAC08</td>
</tr>
<tr>
<td>FPR3</td>
<td>3126E979</td>
</tr>
<tr>
<td>FPR4</td>
<td>40C19400</td>
</tr>
<tr>
<td>FPR5</td>
<td>66666666</td>
</tr>
<tr>
<td>FPR6</td>
<td>3FF661E4F</td>
</tr>
<tr>
<td>FPR7</td>
<td>3FF6666</td>
</tr>
<tr>
<td>FPR8</td>
<td>3FF333333</td>
</tr>
<tr>
<td>FPR9</td>
<td>00000000</td>
</tr>
<tr>
<td>FPR10</td>
<td>98125022</td>
</tr>
<tr>
<td>FPR11</td>
<td>80007F98</td>
</tr>
<tr>
<td>FPR12</td>
<td>40260000</td>
</tr>
<tr>
<td>FPR13</td>
<td>40220000</td>
</tr>
<tr>
<td>FPR14</td>
<td>40220000</td>
</tr>
<tr>
<td>FPR15</td>
<td>40220000</td>
</tr>
</tbody>
</table>

GPREG STORAGE:
Storage around GPR0 (183F6EC0)

**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. A sample output is shown.
Sample Language Environment dump with XPLINK-specific information

The programs `tranmain` shown in Figure 53 on page 202 and `trandll` shown in Figure 54 on page 203 were used to produce a Language Environment dump. 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 sidedeck from the other. The Language Environment dump produced by running these program is shown in Figure 55 on page 204. Explanations for some of the sections are in "Finding XPLINK information in a Language Environment dump" on page 206.
Figure 53. Sample XPLINK-compiled program (tranmain) which calls a NOXPLINK-compiled program
```c
#include <stdio.h>
#include <ctest.h>
#include <leawi.h>
#pragma export(tran1)
#pragma export(tran3)

int tran2(int, int, int, long double, int);

int tran1(int parm1,int parm2,int parm3,long double parm4,int parm5) {
    int retval;
    printf("Tran1: Call Tran2\n");
    retval = tran2(parm1,parm2,parm3,parm4,parm5);
    printf("Tran1: Return value from Tran2 = \%d\n",retval);
    return retval;
}

int tran3(int parm1,int parm2,int parm3,long double parm4,int parm5) {
    _INT4 code, timing;
    code = 1001; /* Abend code to issue */
    timing = 1;
    printf("Tran3: About to ABEND\n");
    CEE3ABD(&code,&timing);
    return parm1 + parm2 + parm3;
}
```

Figure 54. Sample NOXPLINK-compiled program (trandll) which calls an XPLINK-compiled program
Figure 55. Example dump of calling between XPLINK and non-XPLINK programs (Part 1 of 3)
Figure 55. Example dump of calling between XPLINK and non-XPLINK programs (Part 2 of 3)
Finding XPLINK information in a Language Environment dump

[1] Traceback

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().

[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 a 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).

[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 four C/C++ trace table entry types that contain character data:

- Trace entry 1 occurs when a base C library function is called.
- Trace entry 2 occurs when a base C library function returns.
- Trace entry 3 occurs when a POSIX C library function is called.
- Trace entry 4 occurs when a POSIX C library function returns.
- Trace entry 5 occurs when an XPLINK base C or POSIX C library function is called.
- Trace entry 6 occurs when an XPLINK base C or POSIX C library function returns.
- Trace entry 7 occurs when an XPLINK function calls a non-XPLINK function.
- Trace entry 8 occurs when a non-XPLINK function calls an XPLINK function.

The format for trace table entry 1 is:

```
NameOfCallingFunction
-->(xxx) NameOfCalledFunction
```

or, for called functions calloc, free, malloc, and realloc:

```
NameOfCallingFunction
-->(xxx) NameOfCalledFunction<input_parameters>
```

In addition, when the call is due to one of these C++ operators:
then the C++ operator will appear and the format becomes:

```c
NameOfCallingFunction
-->xxx NameOfCalledFunction<\{input_parameters\}>
```

The format for trace table entry 2 is:

```c
<--xxx R15=value ERRNO=value
```

The format for trace table entry 3 is:

```c
NameOfCallingFunction
-->xxx NameOfCalledFunction
```

The format for trace table entry 4 is:

```c
<--xxx R15=value ERRNO=value ERRNO2=value
```

The format for trace table entry 5 is:

```c
NameOfCallingFunction
-->xxx NameOfCalledFunction<\{input_parameters\}>
```

5 is just like trace table entry 1. The `input_parameters` and NameOfC++Operator only appear for the appropriate functions. The angle brackets (`<>`) indicate that this information does not always appear.

The format for trace table entry 6 is:

```c
<--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:

```c
ModuleNameOfCallingFunction:NameOfCallingXplinkFunction
-->ModuleNameOfCalledFunction:NameOfCalledNonXplinkFunction
```

The format for trace table entry 8 is:

```c
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.
Chapter 4. Debugging C/C++ routines
<table>
<thead>
<tr>
<th>Time (00:08:26)</th>
<th>Thread ID</th>
<th>Flags</th>
<th>Entry Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+004080</td>
<td>20.52.46.673373</td>
<td>03</td>
<td>00000000</td>
<td>Isetup</td>
</tr>
<tr>
<td>+004124</td>
<td>20.52.46.673379</td>
<td>03</td>
<td>00000000</td>
<td>Isetup</td>
</tr>
<tr>
<td>+005548</td>
<td>20.52.46.673341</td>
<td>03</td>
<td>00000000</td>
<td>Isetup</td>
</tr>
<tr>
<td>+006816</td>
<td>20.52.47.553343</td>
<td>03</td>
<td>00000000</td>
<td>Isetup</td>
</tr>
<tr>
<td>+007076</td>
<td>20.52.47.553366</td>
<td>03</td>
<td>00000000</td>
<td>Isetup</td>
</tr>
<tr>
<td>+007079</td>
<td>20.52.47.553366</td>
<td>03</td>
<td>00000000</td>
<td>Isetup</td>
</tr>
</tbody>
</table>

Figure 56. Trace table with C/C++ trace table entry types 1 thru 4 (Part 2 of 2)
The following is an example of the format of the trace table entry type 7 and 8:

```
Calculator: calculatethesumoftwointegers
```

Figure 57. Trace table with XPLINK trace table entries 5 and 6.
The following is an example of a dump of the trace table when you specify the LE=20 suboption:

Figure 58. Trace table with trace table entry types 7 and 8 (Part 1 of 2)
Figure 58. Trace table with trace table entry types 7 and 8 (Part 2 of 2)

For more information about the Language Environment trace table format, see "Understanding the trace table entry (TTE)" on page 154.
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 59 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 59. 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 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'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 60 on page 215 shows the generated traceback from the dump.
2. Locate the instruction with the divide-by-zero error in the Pseudo Assembly Listing in Figure 61 on page 216.

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 177.

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 62 shows the WSA address.

Enclave Control Blocks:

::

WSA address..................20914F50
::

Figure 62. C/C++ CAA information in dump

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 63. In this example, the offset is X'0'.

::

---------------------
CLASS C_WSA
LENGTH = AC ATTRIBUTES = MRG, DEFER, RMODE=ANY
OFFSET = 0 IN SEGMENT 002 ALIGN = DBLWORD
---------------------

CLASS
OFFSET NAME TYPE LENGTH SECTION
0 statint PART 4 statint
8 fa PART 4 fa
10 environ PART 4 environ
18 errno PART 4 errno
::

Figure 63. Writable static map

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 64 on page 218.

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 65 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 66 on page 219. 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’+0000005A’ within function funca. The negative offset indicates that the offset cannot be used to locate the instruction that caused the error. Another indication of bad data is the value of X’80000002’ in the instruction

---

Figure 64. Enclave storage section of dump

---

Figure 65. C/C++ example of calling a nonexistent subroutine

---

#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;
}
address of the PSW. This address indicates that an instruction in the routine branched outside the bounds of the routine.

Figure 66. Sections of the dump from example C routine

2. Find the branch instructions at offset X'+0000005A' of funca in the listing in Figure 67 on page 220. 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 68, 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 67. Pseudo assembly listing

Figure 68. Writable static map

Figure 69 on page 221 shows the sections of the dump.
Calling a nonexistent XPLINK function

Figure 70 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.

```c
#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 70. C/C++ example of calling a nonexistent subroutine

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 71 on page 222. 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'+000001C'. 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.

CE3DMP V1 R9.0: Condition processing resulted in the unhandled condition.  01/15/07 1:41:48 PM  Page: 1
ASID: 0051  Job ID: J0B06606  Job name: XEXIST  Step name: STEP1  UserID: HEALY

CEE3845I CEE DUMP Processing started.

Information for enclave main

Information for thread 8000000000000000

Traceback:

<table>
<thead>
<tr>
<th></th>
<th>E Addr</th>
<th>PU Addr</th>
<th>PU Offset</th>
<th>Comp Date</th>
<th>Compile Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2110C500</td>
<td>20902B08</td>
<td>20902B08</td>
<td>20061215</td>
<td>CEL</td>
</tr>
<tr>
<td>2</td>
<td>2110C340</td>
<td>209E0B80</td>
<td>209E0B80</td>
<td>20061215</td>
<td>CEL</td>
</tr>
<tr>
<td>3</td>
<td>21185620</td>
<td>2090158</td>
<td>2090158</td>
<td>2000404</td>
<td>C/C++ XPLINK EBCDIC HFP</td>
</tr>
<tr>
<td>4</td>
<td>21185600</td>
<td>20900000</td>
<td>20900000</td>
<td>2000404</td>
<td>C/C++ XPLINK EBCDIC HFP</td>
</tr>
<tr>
<td>5</td>
<td>21185720</td>
<td>204CA1E0</td>
<td>204CA1B8</td>
<td>20061215</td>
<td>CEL XPLINK EBCDIC HFP</td>
</tr>
<tr>
<td>6</td>
<td>2110C0F0</td>
<td>20C36CE8</td>
<td>20C36CE8</td>
<td>20061214</td>
<td>LIBRARY</td>
</tr>
<tr>
<td>7</td>
<td>2110C030</td>
<td>209A0AD8</td>
<td>209A0AD8</td>
<td>20061215</td>
<td>CEL</td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Condition Information for (DSA address 21185620)

CIB Address: 2110CE20

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:
ILC.... 0002 Interruption Code..... 0001
PSK..... 07002A40 00000002
GPREG STORAGE:
Storage around GPRO (2110C500) -0020 2110C500 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | .................. |
+0000 2110C500 0800CE01 2110C340 2110F620 A09D63A A09EFD08 2110C500 2110C958 20912648 | ...C ...6 ........Q..E...I..j.. |
+0020 2110C520 00000080 20907734 A091500 2090C2A8 2110CE20 A09D665A 2110E4FE 2110D4FF | .................. |

Figure 71. Sections of the dump from example C routine (Part 1 of 2)
2. Find the branch instruction at offset X'+0000001C' of \texttt{funca} in the listing in Figure 72. This instruction is \texttt{BASR r7,r6}. This branch is part of the source statement \texttt{*aa = func_ptr();}.

---

3. Find the offset of \texttt{func_ptr} in the Writable Static Map, shown in Figure 73 on page 224.
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.

```
<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>C_WSA</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 73. Writable static map

Handling dumps written to the z/OS UNIX file system

When a z/OS UNIX C/C++ application program is running in an address space created as a result of a call to `spawn()`, `vfork()`, or one of the `exec` family of functions, the SYSMDUMP DD allocation information is not inherited. Even though the SYSMDUMP allocation is not inherited, a SYSMDUMP allocation must exist in the parent in order to obtain a HFS storage dump.

Alternatively, you can specify the `DYNDUMP` 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/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 83.

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 Panel]

Figure 75. 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 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:

- 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.

Details on how to request and interpret the reports are provided in the following sections.

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. For a sample storage report showing HEAPPOOLS statistics for a multithreaded C/C++ application, see Figure 3 on page 14.

The following describes 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.

Usage 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:
  Initial Heap Size * (Extent Percent/100) / (Element Size)
  with a minimum of four cells.

  **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.
  In order 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.

  **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:
  (Maximum Cells Used * (Element Size) * 100) / 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 will be used for the last suggested cell size.

For more information about stack and heap storage, see **z/OS Language Environment Programming Guide**

---

**C function __uheapreport() storage report**

To generate a user-created heap storage report use the C function, `__uheapreport()`. Use the information in the report to assist with tuning your application’s use of the user-created heap.

**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

---

Chapter 4. Debugging C/C++ routines 227
- \( yyyy \) — the high side of the range
- \( n \) — the number of requests in the range.

- Requests greater than the largest cell size — the number of storage requests that are not satisfied by heap pools.

**Note:** Values displayed in the HeapPools Statistics report are not serialized when collected, therefore the values are not necessarily exact.

**HeapPools summary**

The HeapPools Summary displays a report of the HeapPool Statistics and provides suggested percentages for current cell sizes as well as suggested cell sizes.

- Cell Size — the size of the cell specified on the \texttt{__ucreate()} call
- Extent Percent — the cell pool percent specified on the \texttt{__ucreate()} call
- Cells Per Extent — the number of cells per extent. This number is calculated using the following formula:

\[
\text{Initial Heap Size} \times (\text{Extent Percent}/100)/(8 + \text{Cell Size})
\]

with a minimum of four cells.
- Extents Allocated — the number of times that each pool allocated an extent.
- Maximum Cells Used — the maximum number of cells used for each pool.
- Cells In Use — the number of cells that were never freed.

**Note:** A large number in this field could indicate a storage leak.

- Suggested Percentages for current Cell Sizes — percentages calculated to find the optimal size of the cell pool extent. The calculation is based on the following formula:

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

With a minimum of 1\% and a maximum of 90\%

Make sure that your cell pool extents are neither too large nor too small. If your percentages are too large then additional, unreferenced virtual storage will be allocated, thereby causing the program to exhaust the region size. If the percentages are too small then the HeapPools algorithm will run inefficiently.

- Suggested Cell Sizes — sizes that are calculated to optimally use storage (assuming that the application will \texttt{__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](z/OS Language Environment Programming Guide).

For more information on the \texttt{__uheapreport()} function, see [z/OS XL C/C++ Run-Time Library Reference](z/OS XL C/C++ Run-Time Library Reference). For tuning tips, see [z/OS Language Environment Programming Guide](z/OS Language Environment Programming Guide).

A sample storage report generated by \texttt{__uheapreport()} is shown in Figure 76 on page 229.
The MEMCHECK VHM memory leak analysis tool is an alternative vendor heap manager used to diagnose memory problems. MEMCHECK VHM performs the following functions:

- check for heap storage leaks, double free, and overlays
- trace user heap storage allocation and deallocation requests

The results are displayed in two reports.

**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 the environment variable, _CEE_ENVFILE or _CEE_ENVFILE_S. The format is

CEE_HEAP_MANAGER=dllname

There are two DLLs associated with MEMCHECK VHM:

CEL4MCHK
  31-bit base and xplink.

CELMQMCHK
  64-bit.

They use the following events:

_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.

_VHMREPORT
  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_ENV_FILE 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 once 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 MAX_CALL_LEVELS (100). If an invalid value is specified, the default value 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 4: the minimum is 4 and the maximum is MAX_OVERLAY_LENGTH (80). Non-multiples of 4 will be rounded up to the next multiple.

Default: 4

_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 MAX_CALL_LEVELS (100). If an invalid value is specified, 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. Export _CEE_MEMCHECK_DEPTH=8
   specifies the depth to trace on storage requests (written to the Heap Leak Report)
2. Export _CEE_MEMCHECK_TRACE=ON
   activates the Trace Report option
3. Export _CEE_MEMTRACE_DEPTH=8
   specifies the depth to trace on storage requests (written to the Trace Report)
4. Export _CEE_MEMCHECK_OVERLAY=ON
   activates the Overlay analysis option
5. Export _CEE_HEAP_MANAGER=CEL4MCHK
   activates the tool with the 31-bit DLL (automatically generating the Heap Leak Report)

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 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 0x25a2e0a0
- sequence 12
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 257f6888 +000002b0 _cterm
  Called from: 05d46788 +0000040c (unknown)

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

DEALLOCATE of storage at 0x25a2ecf8
- sequence 10
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25601ae8 +00000084 function3
  Called from: 25601bb8 +0000008c function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000062 main

ALLOCATE of storage at 0x25a2ecf8 for 5 bytes
- sequence 9
  Called from: 25a43c30 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25601ae8 +0000008c function3
  Called from: 25601b88 +0000008c function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a0 +00000062 main

ALLOCATE of storage at 0x25a2ecd8 for 8 bytes
- sequence 8
  Called from: 25a4330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25601bc8 +0000007e function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a0 +00000062 main

DEALLOCATE of storage at 0x25a2ecd8
- sequence 7
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25601c68 +000000cb function1
  Called from: 25601a0 +00000062 main

DEALLOCATE of storage at 0x25a2ecd8
- sequence 6
  Called from: 25a43c78 +00000120 MemFree
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25601c68 +000000ca function1
  Called from: 25601a0 +00000062 main

ALLOCATE of storage at 0x25a2ecd8 for 4 bytes
- sequence 5
  Called from: 25a4330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25601c68 +0000007e function1
  Called from: 25601a0 +00000062 main

ALLOCATE of storage at 0x25a2ec90 for 48 bytes
- sequence 4
  Called from: 25a4330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25725c08 +000000a0 dllinit
  Called from: 05d49c88 +0000007c (unknown)

Figure 77. Trace report generated by MEMCHECK VHM (Part 1 of 2)
The Heap Leak Report 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.

ALLOCATE of storage at 0x25a2ea30 for 584 bytes
- sequence 3
  Called from: 25a44330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPSTFN
  Called from: 258c6d70 +00000186 setlocale
  Called from: 25862540 +0000059e tzset
  Called from: 257fbd30 +00002df2 _cinit
  Called from: 05d4ab00 +00000cb4 (unknown)

ALLOCATE of storage at 0x25a2e1f8 for 2074 bytes
- sequence 2
  Called from: 25a44330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPSTFN
  Called from: 258c6958 +00000070 realloc_name_buffer
  Called from: 258c6d70 +00000132 setlocale
  Called from: 25862540 +0000059e tzset
  Called from: 257fbd30 +00002df2 _cinit
  Called from: 05d4ab00 +00000cb4 (unknown)

ALLOCATE of storage at 0x25a2e0c8 for 280 bytes
- sequence 1
  Called from: 25a44330 +000000fc MemAlloc
  Called from: 05cd9918 +0000005c CEEPSTFN
  Called from: 258c6d70 +00000186 setlocale
  Called from: 25862540 +0000059e tzset
  Called from: 257fbd30 +00002df2 _cinit
  Called from: 05d4ab00 +00000cb4 (unknown)

Figure 77. Trace report generated by MEMCHECK VHM (Part 2 of 2)

The Heap Leak Report 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.
MEMCHECK
Language Environment V1 R7
HEAP LEAK REPORT for enclave main, termination report

Total number of ALLOCATE calls = 7
Total number of DEALLOCATE calls = 5

Current number of bytes allocated = 288928
Maximum number of bytes allocated = 289824

Total number of unmatched ALLOCATE calls = 3
Unmatched ALLOCATE of 8 bytes at address 0x25a2ecd8
- sequence 8
  Called from: 25a44330 +000000d2 MemAlloc
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25601bb8 +0000007e function2
  Called from: 25601c68 +000000ca function1
  Called from: 25601a60 +00000062 main

Unmatched ALLOCATE of 40 bytes at address 0x25a2ec90
- sequence 4
  Called from: 25a44330 +000000d2 MemAlloc
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25725c08 +000000a0 dllinit
  Called from: 05d49c88 +0000007c (unknown)

Unmatched ALLOCATE of 2074 bytes at address 0x25a2e1f8
- sequence 2
  Called from: 25a44330 +000000d2 MemAlloc
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 258c6958 +00000070 realloc_name_buffer
  Called from: 258c6d70 +00000132 setlocale
  Called from: 259e2540 +0000059e tzset
  Called from: 257f8d30 +00002df2 _cinit
  Called from: 05d4abb0 +00000cb4 (unknown)

Total number of unmatched DEALLOCATE calls = 1
Unmatched DEALLOCATE at address 0x25a2ecd8
- sequence 7
  Called from: 25a43c78 +000000f2 MemFree
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25601c68 +000000bc function1
  Called from: 25601a60 +00000062 main

Total number of OVERLAY calls = 1
OVERLAY damage using more than 5 bytes requested at address 0x25a2ecf8
Called from: 25a44330 +000000d2 MemAlloc
Called from: 05cd9918 +0000005c CEEPGTFN
Called from: 25601ae8 +00000084 function3
Called from: 25601bb8 +0000008c function2
Called from: 25601c68 +000000ca function1
 Called from: 25601a60 +00000062 main

Figure 78. Heap Leak Report generated by MEMCHECK VHM

Note: 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 chapter 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.

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.

For example, to check logic flow, you might insert:

```
DISPLAY "ENTER CHECK PROCEDURE".

. (checking procedure routine)
.
DISPLAY "FINISHED CHECK PROCEDURE".
```

To determine whether 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. For a detailed description of the DISPLAY statement, see [Enterprise COBOL for z/OS Language Reference, SC23-8528 or Enterprise COBOL for z/OS Language Reference, SC27-1408].

Scope terminators can also help you trace the logic of your program because they clearly indicate the end of a statement. For a detailed description of scope terminators, see

- [Enterprise COBOL for z/OS Programming Guide, SC23-8529]
- [Enterprise COBOL for z/OS Programming Guide, SC27-1412]
- [COBOL for OS/390 & VM Programming Guide]
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.

The file status key values and their associated meanings are described in the following documents:

- Enterprise COBOL for z/OS Language Reference, SC23-8528
- Enterprise COBOL for z/OS Language Reference, SC27-1408
- COBOL for OS/390 & VM Language Reference

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. See the rules for coding such usage statements in the following documents:

- Enterprise COBOL for z/OS Language Reference, SC23-8528
- Enterprise COBOL for z/OS Language Reference, SC27-1408
- COBOL for OS/390 & VM Language Reference

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. For a detailed discussion of how to use the class test to check for incompatible data, see

- Enterprise COBOL for z/OS Programming Guide, SC23-8529
- Enterprise COBOL for z/OS Programming Guide, SC27-1412
- COBOL for OS/390 & VM Programming Guide

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 whether or not the uninitialized switch is the cause of the problem. For a detailed discussion of how to use the INITIALIZE and SET statements, see

- Enterprise COBOL for z/OS Programming Guide, SC23-8529
- Enterprise COBOL for z/OS Programming Guide, SC27-1412
- COBOL for OS/390 & VM Programming Guide
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.

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.

Code each USE FOR DEBUGGING declarative in a separate section in the DECLARATIVES SECTION of the PROCEDURE DIVISION. See the rules for coding them in the following documents:

- Enterprise COBOL for z/OS Language Reference SC23-8528
- Enterprise COBOL for z/OS Language Reference SC27-1408
- COBOL for OS/390 & VM Language Reference

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 79 on page 240 shows how to use the DISPLAY statement and the USE FOR DEBUGGING declarative to debug a program.
In the example in Figure 79, 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:

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

message every time the PERFORM 501-SOME-Routine runs. The total shown, \( nn \), is the value accumulated in the data item named TOTAL.

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:

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

and dropping the word TOTAL from the DISPLAY statement.

---

Figure 79. Example of using the WITH DEBUGGING MODE clause

In the example in Figure 79 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:

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

message every time the PERFORM 501-SOME-Routine runs. The total shown, \( nn \), is the value accumulated in the data item named TOTAL.

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:

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

and dropping the word TOTAL from the DISPLAY statement.
Using COBOL listings

When you are debugging, you can use one or more of the following listings:

- Sorted Cross-Reference listing
- Data Map listing
- Verb Cross-Reference listing
- Procedure Division Listings

This section gives an overview of each of these listings and specifies the compiler option you use to obtain each listing. For a detailed description of available listings, sample listings, and a complete description of COBOL compiler options, see the following documents:

- **Enterprise COBOL for z/OS Programming Guide** SC23-8529
- **Enterprise COBOL for z/OS Programming Guide** SC27-1412
- **COBOL for OS/390 & VM Programming Guide**

<table>
<thead>
<tr>
<th>Table 16. Compiler-generated COBOL listings and their contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>----------------------------------------</td>
</tr>
<tr>
<td>Sorted Cross-Reference Listings</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Data Map listing</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Verb Cross-Reference listing</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Procedure Division listings</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Procedure Division listings</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Generating a Language Environment dump of a COBOL program

The two sample programs shown in Figure 80 on page 242 and Figure 81 on page 243 generate Language Environment dumps with COBOL-specific information.

COBOL program that calls another COBOL program

In this example, 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 81 on page 243, 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 82 on page 244. 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. 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 245.

---

CBL TEST(STMT,SYM),RENT
IDENTIFICATION DIVISION.
　PROGRAM-ID. COBDUMP2.
　AUTHOR. USER NAME

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
　FILE-CONTROL.
　　SELECT OPTIONAL IOFSS1 ASSIGN AS-ESDS1DD
　　ORGANIZATION SEQUENTIAL ACCESS SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD IOFSS1 GLOBAL.
　1 IOFSSIR PIC X(40).

WORKING-STORAGE SECTION.
01 TEMP4.
　　05 A-1 OCCURS 2 TIMES.
　　10 A-2 OCCURS 2 TIMES.
　　15 A-3V PIC X(3).
　　15 A-6 PIC X(3).
77 DMPTITLE PIC X(80).
77 OPTIONS PIC X(255).
77 FC PIC X(12).

LINKAGE SECTION.
01 SALARY-RECORD.
　02 NAME PIC X(10).
　02 DEPT PIC 9(4).
　02 SALARY PIC 9(6).

PROCEDURE DIVISION USING SALARY-RECORD.
START-SEC.
　DISPLAY "STARTING TEST COBDUMP2"
　MOVE "COBOL DUMP" TO DMPTITLE.
　MOVE "XXX" TO A-6(1, 1).
　MOVE "YYY" TO A-6(1, 2).
　MOVE "ZZZ" TO A-6(2, 1).
　MOVE " BLOCKS STORAGE PAGE(55) FILES" TO OPTIONS.
　CALL "CEE3DMP" USING DMPTITLE, OPTIONS, FC.
　DISPLAY "END OF TEST COBDUMP2"
　GOBACK.
END PROGRAM COBDUMP2.
Registers on entry to CEE3DMP:

PM........ 0000
GPR0...... 114808BC GPR1...... 114842C0 GPR2...... 114A4340 GPR3...... 11202BBC
GPR4...... 11202B18 GPR5...... 11480100 GPR6...... 00000000 GPR7...... 00FD100
GPR8...... 114A41D8 GPR9...... 11480040 GPR10...... 11202908 GPR11...... 11202A04
GPR12...... 112129C0 GPR13...... 114841D0 GPR14...... 9120C78 GPR15...... 912EF98
FPR0...... 00000000 00000000 FPR2...... 00000000 00000000
FPR4...... 00000000 00000000 00000000 00000000
GPREG:

Storage around GPR0 (114808BC):
-0020 114808BC 00000000 00000000 00000000 00000000 00000000 00000000 1144A158 114A41D8 00000000 0001E038 |..................&...........
+0000 114808BC 00000000 00000000 00000000 00000000 00000000 00000000 1144A158 114A41D8 00000000 0001E038 |..................&...........

Information for enclave COBDUMP1

Information for thread 0000000000000000

Traceback:

DSA Entry E Offset Statement Load Mod Program Unit Service Status
1 COBDUMP2 +00000496 40 GO COBDUMP2 Call
2 COBDUMP1 +0000034A 23 GO COBDUMP1 Call

Parameters, Registers, and Variables for active routines:

COBDUMP2 (DSA address 114841D0):

UPSTACKDSA
Saved registers:
GPR0...... 114808BC GPR1...... 114842C0 GPR2...... 114A4340 GPR3...... 11202BBC
GPR4...... 11202B18 GPR5...... 11480100 GPR6...... 00000000 GPR7...... 00FD100
GPR8...... 114A41D8 GPR9...... 11480040 GPR10...... 11202908 GPR11...... 11202A04
GPR12...... 112129C0 GPR13...... 114841D0 GPR14...... 9120C78 GPR15...... 912EF98
GPREG:

Storage around GPR0 (114808BC):
-0020 114808BC 00000000 00000000 00000000 00000000 00000000 00000000 1144A158 114A41D8 00000000 0001E038 |..................&...........
+0000 114808BC 00000000 00000000 00000000 00000000 00000000 00000000 1144A158 114A41D8 00000000 0001E038 |..................&...........

Local variables:
13 IFSS1
14 01 IFSSIR X(40) DISP
17 01 TEMP4 AN-GR
18 02 A-1 AN-GR OCCURS 2
19 03 A-2 AN-GR OCCURS 2

Figure 82. 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 83 on page 246, 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 84 on page 247, 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 84. Storage for active COBOL programs (Part 1 of 2)
Enclave-level data

The Enclave Control Blocks section of the dump, shown in Figure 85 on page 249, 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.
Figure 85. Enclave-level data for COBOL programs
**Process-level data**

The Process Control Block section of the dump, shown in Figure 86, displays COBOL process-level control blocks THDCOM, COBCOM, COBVEC, and ITBLK.

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.

COBOL control blocks THDCOM, COBCOM, COBVEC and ITBLK are dumped for IBM service personnel use.

---

**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 87 on page 251 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 88 on page 252.
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 88 on page 252.** 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.**

---

**Figure 87. COBOL example of moving a value outside an array range**

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 88 on page 252.** 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.**
Figure 88. 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 89. This statement moves the 1 value to the array SLOT (J).

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.

4. Find the values of the local variables in the Parameters, Registers, and Variables for Active Routines section of the traceback, shown in Figure 88 on page 252, 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 90 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 90 shows the program.
To understand the traceback information and debug this program, use the following steps:

1. Locate the error message for the original condition under the Condition Information for Active Routines section of the dump, shown in Figure 91. 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 92 on page 255, 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. 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 93. Notice that the value of SUBNAME with usage DISP, has a value of 'UNKNOWN'. Correct the problem by either changing the subroutine name to one that is defined, or by ensuring that the subroutine is available at compile time.

Figure 92. COBOL Listing for COBOL (Part 2 of 2)

Parameters, Registers, and Variables for Active Routines:

COBOL (DSA address 1147E030):

Saved Registers:

GPR0..... 1147E100 GPR1..... 1147E128 GPR2..... 000197FC GPR3..... 000083F0
GPR4..... 00004428 GPR5..... 11208770 GPR6..... 00000000 GPR7..... 00000000
GPR8..... 0000A3A8 GPR9..... 0000A1B8 GPR10..... 000084FB GPR11..... 0000870C
GPR12.... 000084EC GPR13.... 1147E030 GPR14.... 80008760 GPR15.... 9140A5F8

GPREG STORAGE:

Storage around GPR0 (1147E100):

-0020 1147E1B0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 | .......... | j y j t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t t
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 95 on page 258. 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.
Figure 95. 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 96. This is a call to the assembler routine ASSEMZ3.

4. Check offset +64 in the listing for the assembler routine ASSEMZ3, shown in Figure 97 on page 260. 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, SA22-7832.
5. Check local variables for COBOLZ2 in the Local Variables section of the dump shown in Figure 98. 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.

<table>
<thead>
<tr>
<th>Loc</th>
<th>Object Code</th>
<th>Addr1</th>
<th>Addr2</th>
<th>Stmt</th>
<th>Source Statement</th>
<th>Language Environment Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>47F0 F014</td>
<td>00014</td>
<td>2 ASSEMZ3</td>
<td>CEENTRY MAIN=NO,PPA=MAINPPA</td>
<td>PRINT NOGEN</td>
<td></td>
</tr>
<tr>
<td>000056</td>
<td>4150 092C</td>
<td>0092C</td>
<td>38 LA</td>
<td>5,2348</td>
<td>Low order part of quotient</td>
<td></td>
</tr>
<tr>
<td>00005A</td>
<td>1844</td>
<td>39 SR</td>
<td>4,4</td>
<td>Hi order part of quotient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00005C</td>
<td>5861 0000</td>
<td>00000</td>
<td>40 L</td>
<td>6,0(1)</td>
<td>Get pointer to divisor</td>
<td></td>
</tr>
<tr>
<td>000060</td>
<td>4166 0000</td>
<td>00000</td>
<td>41 LA</td>
<td>6,0(6)</td>
<td>Clear hi bit</td>
<td></td>
</tr>
<tr>
<td>000064</td>
<td>5046 0000</td>
<td>00000</td>
<td>42 D</td>
<td>4,0(6)</td>
<td>Do division</td>
<td></td>
</tr>
<tr>
<td>000068</td>
<td>58FD B0C8</td>
<td>000C8</td>
<td>43 CEETERM</td>
<td>RC=0</td>
<td>Terminate with return code zero</td>
<td></td>
</tr>
<tr>
<td>000080</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51 MAINPPA</td>
<td>CEEPPA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>123, Time Stamp = 2007/02/14 14:59:00</td>
<td>01-CEEPP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>124, Version 1 Release 1 Modification 0</td>
<td>01-CEEPP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>135 CEEOSA</td>
<td>Mapping of the Dynamic Save Area</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>228 CEECAA</td>
<td>Mapping of the Common Anchor Area</td>
</tr>
<tr>
<td>000000</td>
<td>551</td>
<td>END ASSEMZ3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000C8</td>
<td>00000000</td>
<td>552</td>
<td>A(0)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 97. Listing for ASSEMZ3**

5. Check local variables for COBOLZ2 in the Local Variables section of the dump shown in Figure 98. 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 99 on page 261 indicating that the value did pass from COBOLZ1 to COBOLZ2.

**Figure 98. Variables section of Language Environment dump for COBOLZ2**
In the Local Variables section of the dump for program COBOLZ1, shown in Figure 100, 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 99. Listing for COBOLZ2

```
* / COBOLZ2
```

Figure 100. Variables section of Language Environment dump for COBOLZ1

```
Local Variables:
   6  77 D-VAL   9999  COMP    00000
```
Chapter 6. Debugging Fortran routines

This chapter 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

This chapter 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 265.

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)
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

The following table contains a list of the contents of the various compiler-generated listings that you might find helpful when you use information in dumps to debug Fortran programs.

*Table 17. 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

```plaintext
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:

```fortran
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 statement

```fortran
CALL PDUMP(R, R, 5, P, P, 5, Q, Q, 5)
```

should be used to dump the three variables in REAL*4 format. If the statement

```fortran
CALL PDUMP(R, Q, 5)
```

is used, all main storage between R and Q is dumped, which might or might not include P, and could include other variables.

### 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} (a_1, b_1, a_2, b_2,...)
```

- **a** and **b**
  - Variables in the program unit. Each indicates an area of storage to be dumped. Either a or b can represent the upper or lower limit of each storage area.

The dump is always produced in character format. A dump format type (unlike for DUMP/PDUMP) must not be specified.

#### Usage considerations for CDUMP/CPDUMP

A load module can occupy a different area of storage each time it is executed. To ensure that the appropriate areas of storage are dumped, the following conventions should be observed.

If an array and a variable are to be dumped at the same time, a separate set of arguments should be used for the array and for the variable. The specification of limits for the array should be from the first element in the array to the last element. For example, assume that B is a character variable and TABLE is a character array.
of 20 elements. The following call to the storage dump routine could be used to
dump TABLE and B in character format, and stop the program after the dump is
taken:

CALL CDUMP(TABLE(1), TABLE(20), B, B)

**SDUMP subroutine**

The SDUMP subroutine provides a symbolic dump that is displayed in a format
dictated by variable type as coded or defaulted in your source. Data is dumped to
the error message unit. The symbolic dump is created by program request, on a
program unit basis, using CALL SDUMP. Variables can be dumped automatically
after abnormal termination using the compiler option SDUMP. For more information
on the SDUMP compiler option, see *VS FORTRAN Version 2 Programming Guide
for CMS and MVS*.

Items displayed are:

- All referenced, local, named, and saved variables in their Fortran-defined data
  representation
- All variables contained in a static common area (blank or named) in their
  Fortran-defined data representation
- All variables contained in a dynamic common area in their Fortran-defined data
  representation
- Nonzero or nonblank character array elements only
- Array elements with their correct indexes

The amount of output produced can be very large, especially if your program has
large arrays, or large arrays in common blocks. For such programs, you might want
to avoid calling SDUMP.

### Syntax

```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 101 on page 270 shows a sample program calling SDUMP and Figure 102 on page 271 shows the resulting output that is generated. In the main program, the statement

```fortran
EXTERNAL PGM1, PGM2, PGM3
```

makes the address of subprograms PGM1, PGM2, and PGM3 available for a call to SDUMP as follows:

```fortran
CALL SDUMP (PGM1, PGM2, PGM3)
```

This causes variables in PGM1, PGM2, and PGM3 to be printed.

In the subprogram PGM1, the statement

```fortran
EXTERNAL PGM2, PGM3
```

makes PGM2 and PGM3 available. (PGM1 is missing because the call is in PGM1.) The statements

```fortran
CALL SDUMP
CALL SDUMP (PGM2, PGM3)
```

dump variables PGM1, PGM2, and PGM3.
Figure 102 on page 271 shows the resulting output generated by the example in Figure 101.
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.

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 103. Sections of the Language Environment dump

Understanding the Language Environment traceback table

Examine the traceback section of the dump, labeled with [1] in Figure 103 for 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.

Identifying condition information

The section labeled [2] in Figure 103 shows the condition information for the active routines, indicating the program message, program unit name, the statement number, and the offset within the program unit where the error occurred.
Identifying variable information
The local variable section of the dump, shown in the section labeled [3] in Figure 103 on page 272, contains information on all variables and arrays in each program unit in the save area chain, including the program causing 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.

Identifying file status information
The section labeled [4] in Figure 103 on page 272 shows the file status and attribute section of the dump. This section displays the total number of units defined, the default units for error messages, and the default unit numbers for formatted input or formatted output.

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 104 illustrates an error caused by calling a nonexistent routine. The options in effect at compile time appear at the top of the listing.

```fortran
OPTIONS IN EFFECT: LIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NODDIM NORENT SDUMP(ISN) NOSXM NOVECTOR 1L(DIM) NOTEST SC(+) NODE NOEC NOEMODE NOICA NODIRECTIVE NODBS NOSAA NOPARALLEL NODYNAMIC NOSYM NOREORDER NOPC OPT(0) LANGLVL(77) NOFIPS FLAG(I) HALT(S) AUTODBL(NONE) PTRSIZE(8) LINECOUNT(60) CHARLEN(500) NAME(MAIN#)
1      PROGRAM CALLNON
2      CALL NON
3          CALL SUBNAM
4      STOP
5      END
```

Figure 104. Example of calling a nonexistent routine
Figure 105 on page 274 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 Common Lotus Notes Debugging Guide. 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 104 on page 273. 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.
Figure 106 demonstrates a divide-by-zero error. In this example, the main Fortran program passed 0 to subroutine DIVZEROSUB, and the error occurred when DIVZEROSUB attempted to use this data as a divisor.

OPTIONS IN EFFECT: LIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NODIM NORENT SDUMP(ISN)
NOSXM NOVECTOR IL(DIM) NOTEST SC(*) NOEC NOEMODE NOICA NODIRECTIVE NODBCS NOSAA NOPARALLEL NODYNAMIC NOSYM
NOREORDER NOPC
OPT(0) LANGlvl(77) NOFIPS FLAG(I) HALT(S) AUTODBL(NONE) PTRSIZE(8) LINECOUNT(60) CHARLEN(500) NAME(MAIN#)
  1 PROGRAM DIVZERO
  2 INTEGER*4 ANY_NUMBER
  3 INTEGER*4 ANY_ARRAY(3)
  4 PRINT *, 'EXAMPLE STARTING'
  5 ANY_NUMBER = 0
  6 DO I = 1, 3
  7 ANY_ARRAY(I) = I
  8 END DO
  9 CALL DIVZEROSUB(ANY_NUMBER, ANY_ARRAY)
 10 PRINT *, 'EXAMPLE ENDING'
 11 STOP
 12 END

OPTIONS IN EFFECT: LIST NOMAP NOXREF NOGOSTMT NODECK SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NODIM NORENT SDUMP(ISN)
NOSXM NOVECTOR IL(DIM) NOTEST SC(*) NOEC NOEMODE NOICA NODIRECTIVE NODBCS NOSAA NOPARALLEL NODYNAMIC NOSYM
NOREORDER NOPC
OPT(0) LANGlvl(77) NOFIPS FLAG(I) HALT(S) AUTODBL(NONE) PTRSIZE(8) LINECOUNT(60) CHARLEN(500) NAME(MAIN#)
  1 SUBROUTINE DIVZEROSUB(DIVISOR, DIVIDEND)
  2 INTEGER*4 DIVISOR
  3 INTEGER*4 DIVIDEND(3)
  4 PRINT *, 'IN SUBROUTINE DIVZEROSUB'
  5 DIVIDEND(1) = DIVIDEND(3) / DIVISOR
  6 PRINT *, 'END OF SUBROUTINE DIVZEROSUB'
  7 RETURN
  8 END

Figure 106. Fortran routine with a divide-by-zero error

Figure 107 on page 276 shows the Language Environment dump for routine DIVZERO.
Figure 107. Language Environment dump from divide-by-zero Fortran example

To debug this application, do the following:

1. Locate the error message for the current condition in the Condition Information section of the dump, shown in Figure 107. The message is CEE3209S. 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 AFHCLCNNR, which is a Fortran library subroutine.
   b. AFHCLCNNR called DIVZEROSUB.

Figure 107
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.

3. Locate statement 5 in the Fortran listing for the DIVZEROSUB subroutine in Figure 107 on page 276. 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 107 on page 276. 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 chapter 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 part of this chapter 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 the chapter 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. The topics covered are listed below.

- Determining the source of errors in PL/I for MVS & VM routines
- Using PL/I for MVS & VM compiler listings
- Generating a Language Environment dump of a PL/I for MVS & VM routine
- Finding PL/I for MVS & VM information in a dump
- Debugging example of PL/I for MVS & VM routines

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
- Overlaid storage

Logic errors in the source routine

Errors of this type are often difficult to detect because they often appear as compiler or library malfunctions.

Some common errors in source routines are:

- Incorrect conversion from arithmetic data
- Incorrect arithmetic and string manipulation operations
- Unmatched data lists and format lists

Invalid use of PL/I for MVS & VM

A misunderstanding of the language or a failure to provide the correct environment for using PL/I for MVS & VM can result in an apparent malfunction of a PL/I for MVS & VM routine.

Any of the following, for example, might cause a malfunction:

- Using uninitialized variables
- Using controlled variables that have not been allocated
- Reading records into incorrect structures
- Misusing array subscripts
- Misusing pointer variables
- Incorrect conversion
• Incorrect arithmetic operations
• Incorrect string manipulation operations

Unforeseen errors
If an error is detected during run time and no ON-unit is provided in the routine to
terminate the run or attempt recovery, the job terminates abnormally. However, the
status of a routine at the point where the error occurred can be recorded by using
an ERROR ON-unit that contains the statements:

```
ON ERROR
BEGIN;
  ON ERROR SYSTEM;
  CALL PLIDUMP; /*generates a dump*/
  PUT DATA; /*displays variables*/
END;
```

The statement ON ERROR SYSTEM ensures that further errors do not result in a
permanent loop.

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:
    ```
    DCL ARRAY(10);
    :
    DO I = 1 TO 100;
    ARRAY(I) = VALUE;
    ```
To detect this type of error in a compiled module, set the SUBSCRIPTRANGE 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:

```
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:

```
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:

- The use of a subscript outside the declared bounds (check the SUBSCRIPTRANGE condition)
- An attempt to assign a string to a target with an insufficient maximum length (check the STRINGSIZE condition)
- 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)
- 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)
- The reading of a variable-length file into a variable
- The misuse of a pointer variable
- 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.

Invalid calls:
DCL CEEDATE ENTRY OPTIONS(ASM);
CALL CEEDATE(x,y,z); /* invalid */

Valid calls:
DCL CEEDATE ENTRY(*,*,*,* OPTIONAL) OPTIONS(ASM);
CALL CEEDATE(x,y,z,*); /* valid */
CALL CEEDATE(x,y,z,fc); /* valid */

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 below are from the PL/I for MVS & VM product.

Generating PL/I for MVS & VM listings and maps

The following table 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 18. Compiler-generated PL/I for MVS & VM listings and their contents*

<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 18. 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 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 PL/I for MVS & VM listings

Figure 108 shows an example PL/I for MVS & VM routine that was compiled with LIST and MAP:

*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 108. PL/I for MVS & VM routine compiled with LIST and MAP

Figure 109 on page 284 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 109. 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.

Some typical comments you might find in a static storage listing:

<table>
<thead>
<tr>
<th>Comment</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A..xxx</td>
<td>Address constant for xxx</td>
</tr>
<tr>
<td>COMPILER LABEL CL..n</td>
<td>Compiler-generated label n</td>
</tr>
<tr>
<td>CONDITION CSECT</td>
<td>Control section for programmer-named condition</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>Constant</td>
</tr>
<tr>
<td>CSECT FOR EXTERNAL VARIABLE</td>
<td>Control section for external variable</td>
</tr>
<tr>
<td>D..xxx</td>
<td>Descriptor for xxx</td>
</tr>
<tr>
<td>DED..xxx</td>
<td>Data element descriptor for xxx</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
<td>Data descriptor</td>
</tr>
<tr>
<td>ENVB</td>
<td>Environment control block</td>
</tr>
<tr>
<td>FECB..xxx</td>
<td>Fetch control block for xxx</td>
</tr>
<tr>
<td>DCLCB</td>
<td>Declare control block</td>
</tr>
<tr>
<td>FED..xxx</td>
<td>Format element descriptor for xxx</td>
</tr>
</tbody>
</table>

Table 19. Typical comments in a PL/I for MVS & VM static storage listing

Figure 109. Compiler-generated listings from example PL/I for MVS & VM routine (Part 2 of 2)
Table 19. Typical comments in a PL/I for MVS & VM static storage listing (continued)

<table>
<thead>
<tr>
<th>Comment</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>KD...xxx</td>
<td>Key descriptor for xxx</td>
</tr>
<tr>
<td>LOCATOR...xxx</td>
<td>Locator for xxx</td>
</tr>
<tr>
<td>ONCB</td>
<td>ON statement control block</td>
</tr>
<tr>
<td>PICTURED DED...xxx</td>
<td>Pictured data element descriptor for xxx</td>
</tr>
<tr>
<td>PROGRAM ACON</td>
<td>Program address constant</td>
</tr>
<tr>
<td>RD...xxx</td>
<td>Record descriptor for xxx</td>
</tr>
<tr>
<td>SYMBOL TABLE ELEMENT</td>
<td>Symbol table address</td>
</tr>
<tr>
<td>SYMBOL TABLE...xxx</td>
<td>Symbol table for xxx</td>
</tr>
<tr>
<td>SYMTAB DED...xxx</td>
<td>Symbol table DED for xxx</td>
</tr>
<tr>
<td>USER LABEL...xxx</td>
<td>Source program label for xxx</td>
</tr>
<tr>
<td>xxx</td>
<td>Variable with name xxx. If the variable is not initialized, no text appears against the comment. There is also no static offset if the variable is an array (the static offset can be calculated from the array descriptor, if required).</td>
</tr>
</tbody>
</table>

Variable storage map

For automatic and static internal variables, the variable storage map contains the following information:

- PL/I for MVS & VM identifier name
- Level
- Storage class
- Name of the PL/I for MVS & VM block in which it is declared
- Offset from the start of the storage area, in both decimal and hexadecimal form

If the LIST option is also specified, a map of the static internal and external control sections, called the static storage map, is also produced.

Object code listing

The object code listing consists of the machine instructions and a translation of these instructions into a form that resembles assembler and includes comments, such as source program statement numbers.

The machine instructions are formatted into blocks of code, headed by the statement or line number in the PL/I for MVS & VM source program listing. Generally, only executable statements appear in the listing. DECLARE statements are not normally included. The names of PL/I for MVS & VM variables, rather than the addresses that appear in the machine code, are listed. Special mnemonics are used to refer to some items, including test hooks, descriptors, and address constants.

Statements in the object code listing are ordered by block, as they are sequentially encountered in the source program. Statements in the external procedure are given first, followed by the statements in each inner block. As a result, the order of statements frequently differs from that of the source program.

Every object code listing begins with the name of the external procedure. The actual entry point of the external procedure immediately follows the heading comment REAL ENTRY. The subsequent machine code is the prolog for the block,
which performs block activation. The comment `PROCEDURE BASE` marks the end of the prolog. Following this is a translation of the first executable statement in the PL/I for MVS & VM source program.

Following are the comments used in the listing:

*Table 20. Comments in a PL/I for MVS & VM object code listing*

<table>
<thead>
<tr>
<th>Comment</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN BLOCK xxx</td>
<td>Indicates the start of the begin block with label xxx</td>
</tr>
<tr>
<td>BEGIN BLOCK NUMBER n</td>
<td>Indicates the start of the begin block with number n</td>
</tr>
<tr>
<td>CALCULATION OF COMMONED EXPRESSION FOLLOWS</td>
<td>Indicates that an expression used more than once in the routine is calculated at this point</td>
</tr>
<tr>
<td>CODE MOVED FROM STATEMENT NUMBER n</td>
<td>Indicates object code moved by the optimization process to a different part of the routine and gives the number of the statement from which it originated</td>
</tr>
<tr>
<td>COMPILER GENERATED SUBROUTINE xxx</td>
<td>Indicates the start of compiler-generated subroutine xxx</td>
</tr>
<tr>
<td>CONTINUATION OF PREVIOUS REGION</td>
<td>Identifies the point at which addressing from the previous routine base recommences</td>
</tr>
<tr>
<td>END BLOCK</td>
<td>Indicates the end of a begin block</td>
</tr>
<tr>
<td>END INTERLANGUAGE PROCEDURE xxx</td>
<td>Identifies the end of an ILC procedure xxx</td>
</tr>
<tr>
<td>END OF COMMON CODE</td>
<td>Identifies the end of code used in running more than one statement</td>
</tr>
<tr>
<td>END OF COMPILER GENERATED SUBROUTINE</td>
<td>Indicates the end of the compiler-generated subroutine</td>
</tr>
<tr>
<td>END PROCEDURE</td>
<td>Indicates the end of a procedure</td>
</tr>
<tr>
<td>END PROGRAM</td>
<td>Indicates the end of the external procedure</td>
</tr>
<tr>
<td>INITIALIZATION CODE FOR xxx</td>
<td>Indicates the start of initialization code for variable xxx</td>
</tr>
<tr>
<td>INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS</td>
<td>Indicates that some of the code that follows was moved from within a loop by the optimization process</td>
</tr>
<tr>
<td>INTERLANGUAGE PROCEDURE xxx</td>
<td>Identifies the start of an implicitly generated ILC procedure xxx</td>
</tr>
<tr>
<td>METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED</td>
<td>Indicates that the order of the code following was changed to optimize the object code</td>
</tr>
<tr>
<td>ON-UNIT BLOCK NUMBER n</td>
<td>Indicates the start of an ON-unit block with number n</td>
</tr>
<tr>
<td>ON-UNIT BLOCK END</td>
<td>Indicates the end of the ON-unit block</td>
</tr>
<tr>
<td>PROCEDURE xxx</td>
<td>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>
</tbody>
</table>
Table 20. Comments in a PL/I for MVS & VM object code listing (continued)

<table>
<thead>
<tr>
<th>Comment</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<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 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 21. 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:

A All. Results in a dump of all tasks including the ones in the WAIT state.

B BLOCKS (PL/I for MVS & VM hexadecimal dump). Dumps the control blocks used in Language Environment and member language libraries. For PL/I for MVS & VM, this includes the DSA for every routine on the call chain and PL/I for MVS & VM “global” control blocks, such as Tasking Implementation Appendage (TIA), Task Communication Area (TCA), and the PL/I Tasking Control Block (PTCB). PL/I file control blocks and file buffers are also dumped if the F option is specified.

C Continue. The routine continues after the dump.

E Exit. The enclave terminates after the dump. In a multitasking environment, if PLIDUMP is called from the main task, the enclave terminates after the dump. If PLIDUMP is called from a subtask, the subtask and any subsequent tasks created from the subtask terminate after the dump. In a multithreaded environment, if PLIDUMP is called from the Initial Process Thread (IPT), the enclave terminates after the dump. If PLIDUMP is called from a non-IPT, only the non-IPT terminates after the dump.

F FILE INFORMATION. A set of attributes for all open files is given. The contents of the file buffers are displayed if the B option is specified.
**H**  STORAGE in hexadecimal. A SNAP dump of the region is produced. A ddname of CEESNAP must be provided to direct the CEESNAP dump report.

**K**  BLOCKS (when running under CICS). The Transaction Work Area is included.

**Note:** This option is not supported under Enterprise PL/I.

**NB**  NOBLOCKS.

**NF**  NOFILES.

**NH**  NOSTORAGE.

**NK**  NOBLOCKS (when running under CICS).

**NT**  NOTRACEBACK.

**O**  THREAD(CURRENT). Results in a dump of only the current task or current thread (the invoker of PLIDUMP).

**S**  Stop. The enclave terminates after the dump. In a multitasking environment, regardless of whether PLIDUMP is called from the main task or a subtask, the enclave terminates after the dump. In a multithreaded environment, regardless of whether PLIDUMP is called from the IPT or a non-IPT, the enclave terminates after the dump (in which case there is no fixed order as to which thread terminates first).

**T**  TRACEBACK. Includes a traceback of all routines on the call chain. The traceback shows transfers of control from either calls or exceptions. BEGIN blocks and ON-units are also control transfers and are included in the trace. The traceback extends backwards to the main program of the current thread.

T, F, C, and A are the default options.

**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 110 on page 292, 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 111 on page 293.
Figure 111. Task traceback section (Part 1 of 2)
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 112 on page 295 shows this section of the dump.
### Control Blocks for Active Routines:

#### DSA for CEEKMRRA: 204588B

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+00000</td>
<td>FLAGS......</td>
<td>0000</td>
</tr>
<tr>
<td>+00010</td>
<td>R15......</td>
<td>A09B58B</td>
</tr>
<tr>
<td>+00024</td>
<td>R4......</td>
<td>00000001</td>
</tr>
<tr>
<td>+00038</td>
<td>R9......</td>
<td>00000000</td>
</tr>
<tr>
<td>+0004C</td>
<td>NAB......</td>
<td>204588B</td>
</tr>
<tr>
<td>+00064</td>
<td>reserved.</td>
<td>00000000</td>
</tr>
</tbody>
</table>

- | +00078 reserved. | 00000000 |

#### DSA for IBMRERR: 00052670

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+00000</td>
<td>FLAGS......</td>
<td>0000</td>
</tr>
<tr>
<td>+00010</td>
<td>R15......</td>
<td>A09B420</td>
</tr>
<tr>
<td>+00024</td>
<td>R4......</td>
<td>00000006</td>
</tr>
<tr>
<td>+00038</td>
<td>R9......</td>
<td>00000000</td>
</tr>
<tr>
<td>+0004C</td>
<td>NAB......</td>
<td>204588B</td>
</tr>
<tr>
<td>+00064</td>
<td>reserved.</td>
<td>00000000</td>
</tr>
</tbody>
</table>

- | +00078 reserved. | 00000000 |

#### DSA for ERR ON-unit: 20458AB

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+00000</td>
<td>FLAGS......</td>
<td>CC5</td>
</tr>
<tr>
<td>+00010</td>
<td>R15......</td>
<td>A08C164</td>
</tr>
<tr>
<td>+00024</td>
<td>R4......</td>
<td>00000004</td>
</tr>
<tr>
<td>+00038</td>
<td>R9......</td>
<td>00000000</td>
</tr>
<tr>
<td>+0004C</td>
<td>NAB......</td>
<td>204588B</td>
</tr>
</tbody>
</table>

- | +00064 reserved. | 00000000 |

#### Dynamic save area (ERR ON-unit): 2045A5B

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+00000</td>
<td>2045A5B</td>
<td>208A31F</td>
</tr>
<tr>
<td>+00010</td>
<td>900E40</td>
<td>0000064</td>
</tr>
<tr>
<td>+00024</td>
<td>2045A5B</td>
<td>208A31F</td>
</tr>
<tr>
<td>+00038</td>
<td>2045A5B</td>
<td>9001E0D</td>
</tr>
<tr>
<td>+0004C</td>
<td>2045A5B</td>
<td>900E40</td>
</tr>
</tbody>
</table>

- | +00064 reserved. | 00000000 |

---

**Figure 112. Control blocks for active routines section of the dump (Part 1 of 3)**
Figure 112. 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'0C8' 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:
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 113 on page 299, includes information about PL/I for MVS & VM fields of the CAA and other control block information.
Control Blocks Associated with the Thread:

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000000000000025658</td>
<td>20824018 2062018 00000000 2090E900 00000000 02051800</td>
</tr>
<tr>
<td>0000000000000000000000000025659</td>
<td>02052400 00000000 02052800 00000000 02052800 00000000</td>
</tr>
<tr>
<td>000000000000000000000000002565A</td>
<td>00000000 02052A00 00000000 02052B00 00000000 02052D00</td>
</tr>
<tr>
<td>000000000000000000000000002565B</td>
<td>00000000 02052F00 00000000 02053300 00000000 02053500</td>
</tr>
<tr>
<td>000000000000000000000000002565C</td>
<td>00000000 02053700 00000000 02053900 00000000 02053B00</td>
</tr>
<tr>
<td>000000000000000000000000002565D</td>
<td>00000000 02053F00 00000000 02054100 00000000 02054300</td>
</tr>
<tr>
<td>000000000000000000000000002565E</td>
<td>00000000 02054500 00000000 02054700 00000000 02054900</td>
</tr>
<tr>
<td>000000000000000000000000002565F</td>
<td>00000000 02054B00 00000000 02054D00 00000000 02054F00</td>
</tr>
</tbody>
</table>

```

To view the entire block, refer to pg 299.
Figure 113. Control blocks associated with the thread section of the dump (Part 2 of 2)

The CAA
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:

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

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

—>(102) NameOfReturnTask

For more information about the Language Environment trace table format, see “Understanding the trace table entry (TTE)” on page 154.

### 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 114 on page 302 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 on page 303 shows sections of the dump generated by a call to PLIDUMP.

<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EXAMPLE: PROC OPTIONS(MAIN);</td>
</tr>
<tr>
<td>2</td>
<td>DCL Array(10) Fixed bin(31);</td>
</tr>
<tr>
<td>3</td>
<td>DCL (I,Array_End) Fixed bin(31);</td>
</tr>
<tr>
<td>4</td>
<td>On error Begin;</td>
</tr>
<tr>
<td>5</td>
<td>On error system;</td>
</tr>
<tr>
<td>6</td>
<td>Call plidump('tnfns','Plidump called from error On-unit');</td>
</tr>
<tr>
<td>7</td>
<td>End;</td>
</tr>
<tr>
<td>8</td>
<td>(subrg): /* Enable subscriptrange condition */</td>
</tr>
<tr>
<td>9</td>
<td>Labl1: Begin;</td>
</tr>
<tr>
<td>10</td>
<td>Array_End = 20;</td>
</tr>
<tr>
<td>11</td>
<td>Do I = 1 to Array_End; /* Loop to initialize array */</td>
</tr>
<tr>
<td>12</td>
<td>Array(I) = 2; /* Set array elements to 2 */</td>
</tr>
<tr>
<td>13</td>
<td>End;</td>
</tr>
<tr>
<td>14</td>
<td>End Labl1;</td>
</tr>
<tr>
<td>15</td>
<td>End Example;</td>
</tr>
</tbody>
</table>

Figure 114. Example of moving a value outside an array range

<table>
<thead>
<tr>
<th>IDENTIFIER</th>
<th>LEVEL</th>
<th>OFFSET</th>
<th>HEX</th>
<th>CLASS</th>
<th>BLOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
<td>200</td>
<td>C8</td>
<td>AUTO</td>
<td>EXAMPLE</td>
</tr>
<tr>
<td>ARRAY_END</td>
<td>1</td>
<td>204</td>
<td>CC</td>
<td>AUTO</td>
<td>EXAMPLE</td>
</tr>
<tr>
<td>ARRAY</td>
<td>1</td>
<td>208</td>
<td>DO</td>
<td>AUTO</td>
<td>EXAMPLE</td>
</tr>
</tbody>
</table>

5688-235 IBM PL/I for MVS & VM  Ver 1 Rel 1 Mod 1  27 FEB 07  11:45:18  PAGE 1
OPTIONS SPECIFIED
*PROCESS  GOSTMT LIST S STG TEST MAP NOOPTIONS;
5688-235 IBM PL/I for MVS & VM  EXAMPLE: PROC OPTIONS(MAIN);
SOURCE LISTING

...
Figure 115. Sections of the Language Environment dump (Part 1 of 2)
Figure 115. 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.

   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.

2. Locate statement 9 in the routine in Figure 114 on page 302. The instruction is

   Arry_End = 20. This statement assigns a 20 value to the variable Arry_End.

3. Statement 10 begins the DO-loop instruction Do I = 1 to Arry_End. Since the previous instruction (statement 9) specified that Arry_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 114 on page 302. 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 115 on page 303. The block located at this offset contains the value that exceeded the array range, X'B' or 11.

### Calling a nonexistent subroutine

Figure 116 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.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EXAMPLE1: PROC OPTIONS(MAIN);</td>
</tr>
<tr>
<td>2</td>
<td>DCL Prog01 entry external;</td>
</tr>
<tr>
<td>3</td>
<td>On error Begin;</td>
</tr>
<tr>
<td>4</td>
<td>On error system;</td>
</tr>
<tr>
<td>5</td>
<td>Call plidump('tbnfs','Plidump called from error On-unit');</td>
</tr>
<tr>
<td>6</td>
<td>End;</td>
</tr>
<tr>
<td>7</td>
<td>Call prog01; /* Call external program PROG01 */</td>
</tr>
<tr>
<td>8</td>
<td>End Example1;</td>
</tr>
</tbody>
</table>

---

**Figure 116. Example of calling a nonexistent subroutine**

Figure 117 on page 306 shows the traceback and condition information from the dump.
Figure 117. Sections of the Language Environment dump (Part 1 of 2)
Figure 117. Sections of the Language Environment dump (Part 2 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](https://www.ibm.com/support/knowledgecenter/SSEQ00_1.10.0/com.ibm.zos.msc.rsqpe00/rmzsec000.htm).

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’-00900D2C’ 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 116 on page 305). 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 118 on page 308 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 119 shows this output.

```plaintext
Sample Starting
Sub1 Starting
A_NUMBER=0
MY_NAME='Tery Gillaspy'
AN_ARRAY(1)=1
AN_ARRAY(2)=3
AN_ARRAY(3)=5;
```

Figure 119. Variables from routine SAMPLE
The routine in Figure 118 on page 308 was compiled with the LIST compiler option, which generated the object code listing shown in Figure 120.

* STATEMENT NUMBER 15
0003A2 58 80 D 0C8 L 11,200(0,13)
0003A6 58 40 B 004 L 4,4(0,11)
0003AA 58 90 3 0B4 L 9,180(0,3)
0003AE 5C 80 4 004 M 8,4(0,4)
0003B2 58 70 3 004 L 7,212(0,3)
0003B6 5C 60 4 004 M 6,4(0,4)
0003BA 58 80 D 0C0 L 8,192(0,13)
0003BE 58 60 8 000 L 6,0(0,11)
0003C2 5F 60 4 000 SL 6,0(0,4)
0003C6 58 E7 6 000 L 14,VO..ARRAY1(7)
0003CA 8E ED 0 020 SRDA 14,32
0003CE 5D EE 8 000 D 14,DIVISOR
0003D2 50 F9 6 000 ST 15,VO..ARRAY1(9)

Figure 120. Object code listing from example PL/I for MVS & VM routine

Figure 121 on page 310 shows the Language Environment dump for routine SAMPLE.
Figure 121. Language Environment dump from example PL/I for MVS & VM routine (Part 1 of 3)
Control Blocks for Active Routines:

DSA for ERR ON-UNIT: 20845940
+000000 FLAGS.... 0040 member.... BKC.... 20845708.... FWCT.... 04040400.... R14.... 00090304
+000010 R15.... 0081C0A0.... R0.... 20845268 R1.... 20840958 R2.... 000902B8 R3.... 208409A8
+000024 R4.... 000000064 R5.... 00000000 R6.... 20840958 R7.... 208457F0 R8.... 20845240
+000038 R9.... 20845460 R10.... 00000000 R11.... 2084095C R12.... 0001A4FF R13.... 208405C7 reserved. 00052560
+00004C NAB.... 20845268 reserved. 20845268 reserved. 20840563 reserved. 20840563 reserved. 00052560
+000064 reserved. 04040400 reserved. 04040400 MDDE.... 04040404 reserved. 208405F8
+000078 reserved. 04040400 reserved. 04040400

Dynamic save area (ERR_ON-UNIT): 20845940
+000000 20845940 C25240A0 20845708 04040400 09003036 0001C0A0 20845A68 2084059A 000902B8 
... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ......
Figure 121. Language Environment dump from example PL/I for MVS & VM routine (Part 3 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 120 on page 309. 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 chapter 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 chapter 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. The topics covered are listed below.

- Determining the source of errors in Enterprise PL/I routines
- Using Enterprise PL/I compiler listings
- Generating a Language Environment dump of an Enterprise PL/I routine
- Finding Enterprise PL/I information in a dump
- Debugging example of Enterprise PL/I routines

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 statements:

```
ON ERROR
BEGIN;
  ON ERROR SYSTEM;
  CALL PLIDUMP; /*generates a dump*/
  PUT DATA; /*displays variables*/
END;
```

The statement ON ERROR SYSTEM ensures that further errors do not result in a permanent loop.

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:
    ```
    DCL ARRAY(10);
    ...
    DO I = 1 TO 100;
    ARRAY(I) = VALUE;
    ```

    To detect this type of error in a compiled module, set the SUBSCRIPTRANGE 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:

  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:

  DCL X CHAR(3);
  I=3
  SUBSTR(X,2,I) = 'ABC';

  To detect this type of error, enable the STRINGRANGE condition during compilation.

Storage overlay problems

If you suspect an error in your Enterprise PL/I application is a storage overlay problem, check for the following:

- The use of a subscript outside the declared bounds (check the SUBSCRIPTRANGE condition)

- An attempt to assign a string to a target with an insufficient maximum length (check the STRINGSIZE condition)

- 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)

- 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)

- The reading of a variable-length file into a variable

- The misuse of a pointer variable

- 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.

Invalid calls:
DCL CEEDATE ENTRY OPTIONS(ASM);
CALL CEEDATE(x,y,z); /* invalid */

Valid calls:
DCL CEEDATE ENTRY(*,*,*,* OPTIONAL) OPTIONS(ASM);
CALL CEEDATE(x,y,z,*); /* valid */
CALL CEEDATE(x,y,z,fc); /* valid */

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
The following table shows compiler-generated listings that you might find helpful when you use information in dumps to debug Enterprise PL/I routines.

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source program</td>
<td>Source program statements</td>
<td>SOURCE</td>
</tr>
<tr>
<td>Cross reference</td>
<td>Cross reference of names with attributes</td>
<td>XREF and ATTRIBUTES</td>
</tr>
<tr>
<td>Aggregate table</td>
<td>Names and layouts of structures and arrays</td>
<td>AGGREGATE</td>
</tr>
<tr>
<td>Variable map</td>
<td>Offsets of automatic and static internal variables (from their defining base)</td>
<td>MAP</td>
</tr>
<tr>
<td>Object code</td>
<td>Contents of the program control section in hexadecimal notation and translated into a pseudo-assembler format.</td>
<td>LIST</td>
</tr>
</tbody>
</table>
### Table 22. 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 122 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 122. Enterprise PL/I routine compiled with LIST, MAP, and SOURCE

Figure 123 on page 320 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 123. Compiler-generated listings from example Enterprise PL/I routine (Part 1 of 4)
### OFFSET OBJECT CODE | LINE# | FILE# | PSEUDO ASSEMBLY LISTING
--- | --- | --- | ---
0000C2 | A774 0012 | 000011 | JNE 01L3
0000C6 | 4100 6018 | 000011 | LA r0,.Dsc_000002(,r6,24)
0000CA | 4120 6008 | 000011 | LA r2,.EXTR(,r6,8)
0000CE | 58FO 3006 | 000011 | L r15,-V(IBMQRFG)(,r3,6)
0000D2 | 4110 D098 | 000011 | LA r1,MX_TEMP1(,r13,152)
0000D6 | 5020 D098 | 000011 | ST r2,MX_TEMP1(,r13,152)
0000DA | 1B27 | 000011 | LR r5,r7
0000DC | 5020 D09C | 000011 | ST r2,MX_TEMP1(,r13,156)
0000E0 | 5000 D0A0 | 000011 | ST r0,MX_TEMP1(,r13,160)
0000E4 | 05EF | 000011 | BALR r14,r15
0000E6 | | 000011 | $01L3 DS OR
0000E6 | 4110 6008 | 000011 | LA r1,.EXTR(,r6,8)
0000EA | 58FO 1000 | 000011 | L r15,.Shadow2(,r1,0)
0000EE | 4100 D0B4 | 000011 | LA r0,E(,r13,180)
0000F2 | 1B26 | 000011 | LR r2,r6
0000F4 | 4140 D0B8 | 000011 | LA r4,.temp1(,r13,184)
0000F8 | 4150 D0C0 | 000011 | LA r5,.temp2(,r13,192)
0000FC | 4180 D0B0 | 000011 | LA r8,A(,r13,176)
000100 | 4110 D098 | 000011 | LA r1,MX_TEMP1(,r13,152)
000104 | 5080 D098 | 000011 | ST r8,MX_TEMP1(,r13,152)
00010B | 5050 D09C | 000011 | ST r5,MX_TEMP1(,r13,156)
00010C | 5040 D0A0 | 000011 | ST r4,MX_TEMP1(,r13,160)
000110 | 5020 D0A4 | 000011 | ST r2,MX_TEMP1(,r13,164)
000114 | 5000 D0A8 | 000011 | ST r0,MX_TEMP1(,r13,168)
00011B | 05EF | 000011 | BALR r14,r15
00011A | 4120 6010 | 000012 | LA r2,.Dsc_000001(,r6,16)
00011E | 4100 6020 | 000012 | LA r0,.Dsc_000004(,r6,32)
000122 | 4140 6028 | 000012 | LA r4,C(,r6,40)
000126 | 58FO 300E | 000012 | L r15,-V(IBMQRFS)(,r3,14)
00012A | 4110 D098 | 000012 | LA r1,MX_TEMP1(,r13,152)
00012E | 5040 D098 | 000012 | ST r4,MX_TEMP1(,r13,152)
000132 | 5000 D09C | 000012 | ST r0,MX_TEMP1(,r13,156)
000136 | 4100 0000 | 000012 | LA r0,0
00013A | 5000 D0A0 | 000012 | ST r8,MX_TEMP1(,r13,160)
00013E | 5020 D0A4 | 000012 | ST r2,MX_TEMP1(,r13,164)
000142 | 5000 D0B0 | 000012 | ST r0,MX_TEMP1(,r13,168)
000146 | 05EF | 000012 | BALR r14,r15
000148 | | 000013 | $01L1 DS OR
000148 | 58FO 3012 | 000013 | L r15,-V(IBMQRFSH)(,r3,18)
00014C | 05EF | 000013 | BALR r14,r15
00014E | | 000013 | $01L4 DS OR
00014E | Start of Epilog
--- | --- | ---
00014E | 5800 D004 | 000013 | L r13,4(,r13)
000152 | 58EO 000C | 000013 | L r14,12(,r13)
000156 | 9B28 D01C | 000013 | LM r2,r8,28(r13)
00015A | 051E | 000013 | BALR r1,r14
00015C | 0707 | 000013 | Nopr 7
00015E | 0000 | 000013 |
---
000160 | Start of Literals
--- | --- | ---
000160 | 00000000 | 00000000 | $A(EXAMPLE2)
000164 | 00000000 | 00000000 | $V(IBMQRFG)
000168 | 00000000 | 00000000 | $A(B)
00016C | 00000000 | 00000000 | $V(IBMQRFSH)

**Figure 123. Compiler-generated listings from example Enterprise PL/I routine (Part 2 of 4)**
### Figure 123. Compiler-generated listings from example Enterprise PL/I routine (Part 3 of 4)

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>ID</th>
<th>ADDR</th>
<th>LENGTH</th>
<th>NAME</th>
<th>TYPE</th>
<th>ID</th>
<th>ADDR</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE1</td>
<td>SD</td>
<td>1</td>
<td>000000</td>
<td>000228</td>
<td>EXAMPLE2</td>
<td>SD</td>
<td>2</td>
<td>000000</td>
<td>00005C</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>SD</td>
<td>3</td>
<td>000000</td>
<td>000004</td>
<td>B</td>
<td>SD</td>
<td>4</td>
<td>000000</td>
<td>000010</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>LD</td>
<td>0</td>
<td>000048</td>
<td>000001</td>
<td>CEESG011</td>
<td>ER</td>
<td>5</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMQFRG</td>
<td>ER</td>
<td>6</td>
<td>000000</td>
<td></td>
<td>IBMQJDSB</td>
<td>ER</td>
<td>7</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMQFSH</td>
<td>ER</td>
<td>8</td>
<td>000000</td>
<td></td>
<td>CEESTART</td>
<td>ER</td>
<td>9</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>CEIMAIN</td>
<td>SD</td>
<td>10</td>
<td>000000</td>
<td>00000C</td>
<td>IBMPIPLN</td>
<td>ER</td>
<td>11</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>ER</td>
<td>12</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### External Symbol Cross Reference

<table>
<thead>
<tr>
<th>ORIGINAL NAME</th>
<th>EXTERNAL SYMBOL NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE1</td>
<td>EXAMPLE1</td>
</tr>
<tr>
<td>EXAMPLE2</td>
<td>EXAMPLE2</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>EXAMPLE</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>EXAMPLE</td>
</tr>
<tr>
<td>CEESG011</td>
<td>CEESG011</td>
</tr>
<tr>
<td>IBMQFRG</td>
<td>IBMQFRG</td>
</tr>
<tr>
<td>IBMQJDSB</td>
<td>IBMQJDSB</td>
</tr>
<tr>
<td>IBMQFSH</td>
<td>IBMQFSH</td>
</tr>
<tr>
<td>CEESTART</td>
<td>CEESTART</td>
</tr>
<tr>
<td>CEIMAIN</td>
<td>CEIMAIN</td>
</tr>
<tr>
<td>IBMPIPLN</td>
<td>IBMPIPLN</td>
</tr>
</tbody>
</table>
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 (i.e. if from an include file).
• the instruction in mnemonic format.

**External symbol dictionary**
The external symbol dictionary lists all the external symbols generated for this compilation. For each symbol, it also lists its linkage type and size (in hex).

**External symbol cross reference**
The external symbol dictionary cross reference shows for each external symbol the name that will be visible externally to the linker.

**Storage offset listing**
Each line of the storage offset listing contains the following information for each user variable:
• Its name.
• the number of the block in which it was declared.
• the number of the file in which it was declared.
• the number of the line in which it was declared.
• Its class (automatic, static, etc).
• Its location (as appropriate for its class).
• Its byte length in decimal.

This list is sorted by block number and then by name within each block.

**Static map**
Each line of the static storage map contains the following information for each internal static variable:
• Its hexadecimal offset.
• Its byte length in hex.
• Its name.

This list is sorted by the offset of the variables in static.

This list of variables may also include compiler-generated variables.

**Automatic map**
Each line of the automatic storage map contains the following information, grouped by named block, for each automatic variable in that block:
• Its hexadecimal offset.
• Its byte length in hex.
• Its name.

These lists are sorted by the offset of the variables in automatic for each block.

These lists of variables may also include compiler-generated variables.

---

**Generating a Language Environment dump of an Enterprise PL/I routine**

To generate a dump of an Enterprise PL/I routine, you can call either the Language Environment callable service CEE3DMP or PLIDUMP. For information about calling CEE3DMP, see “Generating a Language Environment dump with CEE3DMP” on page 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 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:

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.

T, F, C, and A are the default options.

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:
    
    Snap was successful; snap ID = mnn
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 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 124 on page 328 for condition information about your routine and information about the statement number and address where the exception occurred.
Figure 124. Traceback section of dump

**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 125 on page 330 shows this section of the dump.
Control Blocks for Active Routines:

- **DSA** for EXAMPLE: 11A3A38B
  - DSA for EXAMPLE: 11A3A38B
  - Dynamic save area (EXAMPLE): 11A3A38B

Dynamic save area (EXAMPLE): 11A3A38B

---

**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.

**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.

Control blocks associated with the thread
This section of the dump, shown in Figure 126 on page 332 includes information about Enterprise PL/I fields of the CAA and other control block information.
Figure 126. Control blocks associated with the thread section of the dump (Part 1 of 2)
Figure 126. Control blocks associated with the thread section of the dump (Part 2 of 2)

The CAA

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

The default and declared attributes of all open files

- Some files are open with the WRITE attribute
- Some files are open with the READ attribute
- Some files are open with both attributes

Buffer contents of all file buffers

- The buffer contents for each file are displayed
- The buffer contents include the file attributes

The contents of FCBs, DCBs, DCLCBs, IOCBs, and control blocks for the process or enclave

- The contents of FCBs, DCBs, DCLCBs, IOCBs, and control blocks are displayed
- The contents include the file attributes

Chapter 8. Debugging Enterprise PL/I routines 333
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 is:

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

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

---(102) NameOfReturnTask

For more information about the Language Environment trace table format, see "Understanding the trace table entry (TTE)" on page 154.

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 127 on page 335 illustrates an error caused by an array subscript value outside the declared range. In this example, the declared array value is 10.

This routine was compiled with the options LIST, TEST, GONUMBER, and MAP. It was run with the TERMTHDACT(TRACE) option to generate a traceback for the condition.
Figure 128 on page 336 shows sections of the dump generated by a call to PLIDUMP.
Information for enclave EXAMPLE

Information for thread 8000000000000000

Traceback:

<table>
<thead>
<tr>
<th>DSA</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service Unit</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+0000002E</td>
<td>_Begin_J2.Blk_2</td>
<td>10MPEV11</td>
<td>P078306</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>+00000002</td>
<td>_Begin_J2.Blk_3</td>
<td>10MPEV11</td>
<td>P076426</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>+0000002A</td>
<td>_Begin_J2.Blk_3</td>
<td>10MPEV11</td>
<td>LE198AS</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>+0000004D</td>
<td>CEEV011</td>
<td>10MPEV11</td>
<td>CEEV011</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>+00000112</td>
<td>CEEV011</td>
<td>CEEHDP</td>
<td>D1098</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>+0000170D</td>
<td>CEEPLPKA</td>
<td>CEEHDP</td>
<td>LE198AS</td>
<td>Exception</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>+000000AA</td>
<td>_ON_Begin_7.Blk_2</td>
<td>10MPEV11</td>
<td>LE198AS</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>+00000009</td>
<td>_Begin_J2.Blk_3</td>
<td>10MPEV11</td>
<td>LE198AS</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>+00000002</td>
<td>_Begin_J2.Blk_3</td>
<td>10MPEV11</td>
<td>LE198AS</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>+00000100</td>
<td>_Begin_J2.Blk_3</td>
<td>10MPEV11</td>
<td>LE198AS</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>+0000008D</td>
<td>_Begin_J2.Blk_3</td>
<td>10MPEV11</td>
<td>LE198AS</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>+0000004E</td>
<td>_Begin_J2.Blk_3</td>
<td>10MPEV11</td>
<td>LE198AS</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>+00000102</td>
<td>_Begin_J2.Blk_3</td>
<td>10MPEV11</td>
<td>LE198AS</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>+00000186</td>
<td>_Begin_J2.Blk_3</td>
<td>10MPEV11</td>
<td>LE198AS</td>
<td>Call</td>
<td></td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Program Unit: Entry: _ON_Begin_7.Blk_2: Offset: +0000002D

Storage dump near condition, beginning at location: 114AA9B2

Condition Information for (DSA address 11A3A68B)

CIB Address: 11A3B176

Current Condition:

_0M02815 A prior condition was promoted to the ERROR condition.

Original Condition:

_0M02815 ONCODE=520 The SUBSCRIPTRANGE condition was raised.

Location:

Program Unit: Entry: _ON_Begin_7.Blk_2: Offset: +0000002D

Storage dump near condition, beginning at location: 114AA9B2

+000000 114AA9B2 411D4DB8 505D4DB8 504D0D0C 504D400A 05EFF100 43002016 A70100B8 A7840098 ...q.&.q& ...&. .......x..x.. |
Figure 128. Sections of the Language Environment dump (Part 2 of 2)

To debug this routine, use the following steps:

1. In the dump, PLIDUMP was called by the ERROR ON-unit in statement 9. The traceback information in the dump shows that the exception occurred following statement 16.
Note: In the LE 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 127 on page 335. 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 127 on page 335. Use this offset to find the I value at the time of the dump. In this example, the offset is X'E8'.

2. Now find offset X'E8' from the start of the stack frame for the entry EXAMPLE in Figure 128 on page 336. The block located at this offset contains the value that exceeded the array range, X'B' or 11.

Calling a nonexistent subroutine

Figure 129 on page 339 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 129. Example of calling a nonexistent subroutine

Figure 130 on page 340 shows the traceback and condition information from the dump.
Figure 130. 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 [2/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'+00001A' statement 12 within entry EXAMPLE1.

2. Locate statement 12 in the routine (Figure 129 on page 339). This statement calls subroutine Prog01. The message CEE3201S, which indicates an operations exception, was generated because of an unresolved external reference.

---

**Figure 130. Sections of the Language Environment dump (Part 2 of 2)**

---
3. Check the linkage editor output for error messages.

**Divide-by-zero error**

Figure 131 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.

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 132 shows this output.
The route in Figure 131 on page 342 was compiled with the LIST compiler option, which generated the object code listing shown in Figure 133.

---

**Figure 133.** Object code listing from example Enterprise PL/I routine

**Figure 134 on page 344** shows the Language Environment dump for routine **SAMPLE**.
Figure 134. Language Environment dump from example Enterprise PL/I routine (Part 1 of 2)
Chapter 8. Debugging Enterprise PL/I routines
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](https://www.ibm.com/support/knowledgecenter/en/SSECG2_11.1.0/com.ibm.zos.v1r11.secg2.doc/tg03d966000002382.htm).

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 133 on page 343. Either method shows that divisor was loaded into register 2 (r2) and used as the divisor in a divide operation.

5. You can see from the declaration of SUB1 that divisor is a parameter passed from SAMPLE. Because of linkage conventions, you can infer that register 1 in the SAMPLE save area points to a parameter list that was passed to SUB1. divisor is the first parameter in the list.

6. In the SAMPLE DSA, the R1 value is X'11A3B450'. This is the address of the parameter list, which is located in static storage.

7. Find the parameter list in the stack frame; the address of the first parameter is X'11A3B484' and the value of the first parameter is X'00000000'. Thus, the exception occurred when SAMPLE passed a 0 value used as a divisor in subroutine SUB1.
Chapter 9. Debugging under CICS

This chapter 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 DFHDCNT 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. A sample Language Environment message that appears when an application abends due to an unhandled condition from an EXEC CICS command is:

```
P039UTV9 19910916145313 CEE3250C The System or User ABEND AEI0 was issued.
P039UTV9 19910916145313 From program unit UT9CVERI at entry point UT9CVERIT +00000011E at P039UTV9 19910916145313 at offset address 0006051E.
```

Locating the Language Environment traceback

Under CICS, the Language Environment traceback is written to the CESE transient data queue. Because Language Environment invokes your application routine, the Language Environment routines that invoked your routine appear in the traceback. *Figure 135 on page 348* 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 43.
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.

```
12.34.27 J0805585 IEF450I XCEPII03 GO CEP1103 - ABEND=S000 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. Following is 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 just formats the heading line for each trace point and is laid out in a similar way to CICS trace entries. For example,
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 example:

```plaintext
Figure 136. CICS trace output in the ABBREV format.
```
For more information about the CICS trace, see [CICS Diagnosis Reference](#).
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. Following is 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. Following is 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 = 0001234</td>
<td>System console</td>
<td>Language Environment</td>
</tr>
</tbody>
</table>
Table 24. Finding data when Language Environment abends internally (continued)

<table>
<thead>
<tr>
<th>Output Message</th>
<th>Location</th>
<th>Issued By</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFHAC2236 06/05/91 14:35:24 LE03CC01 Transaction UTV8 abend 4095 in routine UT8CVERI term P021 backout successful.</td>
<td>Transient data queue CSMT</td>
<td>CICS</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 25. 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)
  – CBLPSHPOP(ON|OFF)
  – CHECK(ON|OFF)
  – INFORMSGFILTER(ON|OFF)
  – RPTOPTS(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:

```
DEFINE PROGRAM(CEL4RTO) GROUP(CEE) LANGUAGE(ASSEMBLER) EXECKEY(CICS)
DEFINE MAPSET(CELCLEM) GROUP(CEE)
DEFINE MAPSET(CELCLRH) GROUP(CEE)
DEFINE TRANS(CLER) PROG(CEL4RTO) GROUP(CEE)
```

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

There are several run-time options that 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:

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.

INFO_MSGFILTER 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.

RPT_OPTS 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 361.

STORAGE Specifies that Language Environment initializes all heap and stack storage to a user-specified value.

TERMTHD ACT 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.

For a more detailed discussion of these run-time options, see z/OS Language Environment Programming Reference.

**Determining run-time options in effect**

The run-time options in effect at the time the routine is run can affect routine behavior. Use RPT_OPTS(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 138 on page 361 shows a sample options report.
Controlling storage allocation

The following run-time options control storage allocation:

- **HEAP64**
- **HEAPPOOLS**
- **HEAPPOOLS64**
- **IOHEAP64**
- **LIBHEAP64**
- **STACK64**
- **THREADSTACK64**

**z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode** provides useful tips to assist with the tuning process. Appropriate tuning is necessary to avoid performance problems.

To generate a report of the storage a routine (or more specifically, an enclave) used during its run, specify the RPTSTG(ON) 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 139 on page 362 shows a sample storage report.
Figure 139. 64–bit storage report (Part 1 of 4)
### 24bit Library HEAP statistics:
Initial size: 8192
Increment size: 4096
Total heap storage used (suggested initial size): 0
Successful Get Heap requests: 0
Successful Free Heap requests: 0
Number of segments allocated: 0
Number of segments freed: 0

### 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 (suggested 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 (suggested 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:

<table>
<thead>
<tr>
<th>Pool</th>
<th>Size</th>
<th>Get Requests</th>
<th>Successful Get Heap requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pool 1</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pool 2</td>
<td>32</td>
<td>1</td>
<td>17-24</td>
</tr>
<tr>
<td>Pool 3</td>
<td>128</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pool 4</td>
<td>256</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pool 5.1</td>
<td>1024</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>Pool 5.2</td>
<td>1024</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Pool 5.3</td>
<td>1024</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Successful Get Heap requests: 273-280

### HEAPPOOLS Summary:

<table>
<thead>
<tr>
<th>Cell Size</th>
<th>Element Extent</th>
<th>Cells Per Extents</th>
<th>Maximum Allocated Cells Used</th>
<th>Cells In Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td>10</td>
<td>409</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>40</td>
<td>10</td>
<td>163</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>136</td>
<td>10</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>264</td>
<td>10</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>10</td>
<td>4</td>
<td>57</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>2048</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)`
- `HEAPP(ON,24,,280,,2048,,0)`

---

Figure 139. 64–bit storage report (Part 2 of 4)
### HEAPPOOLS64 Statistics:

<table>
<thead>
<tr>
<th>Pool</th>
<th>Size</th>
<th>Get Requests</th>
<th>Successful Get Heap requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1-8</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>240</td>
<td>9-16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>17-24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25-32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>33-40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>41-48</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>49-56</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>57-64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>65-72</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>73-80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>81-88</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>89-96</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>105-112</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>113-120</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>121-128</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>53</td>
<td>129-136</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>137-144</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>145-152</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>153-160</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>161-168</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>169-176</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>177-184</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>185-192</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>193-200</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>201-208</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>209-216</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>217-224</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>225-232</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>233-240</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>249-256</td>
</tr>
<tr>
<td>5.1</td>
<td>1024</td>
<td>2</td>
<td>281-288</td>
</tr>
<tr>
<td>5.2</td>
<td>1024</td>
<td>2</td>
<td>521-528</td>
</tr>
<tr>
<td>5.3</td>
<td>1024</td>
<td>0</td>
<td>713-720</td>
</tr>
<tr>
<td>6</td>
<td>2048</td>
<td>2</td>
<td>1505-1512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1641-1648</td>
</tr>
<tr>
<td>7</td>
<td>3072</td>
<td>2</td>
<td>2073-2080</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2105-2112</td>
</tr>
<tr>
<td>8</td>
<td>4096</td>
<td>1</td>
<td>3681-3688</td>
</tr>
<tr>
<td>9</td>
<td>8192</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>16384</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>32768</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>65536</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Requests greater than the largest cell size: 0</td>
</tr>
</tbody>
</table>

*Figure 139. 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.

---

**HEAPPOLLS64 Summary:**

<table>
<thead>
<tr>
<th>Specified Element</th>
<th>Cells Per Extent</th>
<th>Maximum Allocated</th>
<th>Cells Used</th>
<th>Cells In Use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cell Size</th>
<th>Element Size</th>
<th>Cells Per Extent</th>
<th>Maximum Allocated</th>
<th>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>0</td>
</tr>
<tr>
<td>128</td>
<td>144</td>
<td>700</td>
<td>0</td>
<td>83</td>
<td>77</td>
</tr>
<tr>
<td>256</td>
<td>272</td>
<td>350</td>
<td>0</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>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>2064</td>
<td>50</td>
<td>1</td>
<td>0</td>
<td>0</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>0</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 139. 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 516.

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:

__cabend() Terminates an enclave using an abend.
__le_cib_get() Returns a pointer to a condition information block (CIB)
associated with a given condition token. The CIB contains
detailed information about the condition.

__set_exception_handler() Activates a routine to handle an exception.
__reset_exception_handler() Removes handling of an exception by any routine.

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

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
• UAIMITM 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,PIE), 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 370).

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.

<table>
<thead>
<tr>
<th>Function</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>

For more information on these APIs, see [z/OS XL C/C++ Programming Guide].

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 140. 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 361 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 26 on page 372 lists common error symptoms, possible causes, and programmer responses.
Table 26. 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 &quot;Interpreting run-time messages&quot; below.</td>
</tr>
<tr>
<td>User abend code &lt; 4000</td>
<td>a) A non-Language Environment abend occurred</td>
<td>See the Language Environment abend codes in z/OS Language Environment Run-Time Messages. Check for a subsystem-generated abend or a user-specified abend.</td>
</tr>
<tr>
<td></td>
<td>b) The assembler user exit requested an abend for an unhandled condition of severity ≥2</td>
<td></td>
</tr>
<tr>
<td>User abend code ≥ 4000</td>
<td>a) Language Environment detected an error and could not proceed</td>
<td>For any abends you receive, read the appropriate explanation listed in the abend codes section of z/OS Language Environment Run-Time Messages.</td>
</tr>
<tr>
<td></td>
<td>b) An unhandled software-raised condition occurred</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) The assembler user exit requested an abend for an unhandled condition of severity 4</td>
<td></td>
</tr>
<tr>
<td>System abend with TRAP(OFF)</td>
<td>Cause depends on type of malfunction</td>
<td>Respond appropriately. See the messages and codes book of the operating system.</td>
</tr>
<tr>
<td>System abend with TRAP(ON)</td>
<td>System-detected error</td>
<td>See the messages and codes 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 from most recent service)</td>
<td>Condition caused by something related to current service</td>
<td>Generate a traceback using cdump() or ctrace().</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 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)”.

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. 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>
</tbody>
</table>

The messages for the various components can be found in [z/OS Language Environment Run-Time Messages](#).

**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 chapter 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 27. TERMTHDACT suboptions, level of information, and destinations

<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of Information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIET</td>
<td>No information</td>
<td>No destination.</td>
</tr>
<tr>
<td>MSG</td>
<td>Message</td>
<td>Stderr</td>
</tr>
<tr>
<td>TRACE</td>
<td>Message and Language Environment dump containing only a traceback</td>
<td>Message goes to stderr. Traceback goes to CEEDUMP file.</td>
</tr>
<tr>
<td>DUMP</td>
<td>Message and complete Language Environment dump</td>
<td>Message goes to stderr. Language Environment dump goes to CEEDUMP file.</td>
</tr>
</tbody>
</table>
Table 27. TERMTHDACT suboptions, level of information, and destinations (continued)

<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of Information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAONLY</td>
<td>SYSMDUMP, SYSABEND dump, or SYSUDUMP depending on the DD card used in the JCL in z/OS. You will get a system dump of your user address space if the appropriate DD statement is used. <strong>Note:</strong> A Language Environment dump is not generated.</td>
<td>Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified.</td>
</tr>
<tr>
<td>UATRACE</td>
<td>Message, Language Environment dump containing only a traceback, and a system dump of the user address space</td>
<td>Message goes to stderr. Traceback goes to CEEDUMP file. Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified.</td>
</tr>
<tr>
<td>UADUMP</td>
<td>Message, Language Environment dump, and SYSMDUMP, SYSABEND dump, or SYSUDUMP depending on the DD card used in the JCL in z/OS.</td>
<td>Message goes to stderr. Language Environment dump goes to CEEDUMP file. Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified.</td>
</tr>
<tr>
<td>UAIMM</td>
<td>Language Environment generates a system dump of the original abend/program interrupt of the user address space. You will get a system dump of your user address space if the appropriate DD statement is used. After the dump is taken by the operating system, Language Environment condition manager continues processing.</td>
<td>Message goes to stderr. User address space dump goes to ddname specified for z/OS.</td>
</tr>
</tbody>
</table>

The TRACE and UATRACE suboptions of TERMTHDACT use these dump options:
- CONDITION
- ENCLAVE(ALL)
- FILES
- FNAME(CEEDUMP)
- GENOPTS
- NOBLOCKS
- NOENTRY
- NOSTORAGE
- STACKFRAME(ALL)
- THREAD(ALL)
- TRACEBACK
- VARIABLES

The DUMP and UADUMP suboptions of TERMTHDACT use these dump options:
- BLOCKS
- CONDITION
- ENCLAVE(ALL)
- FILES
Considerations for setting TERMTHDACT options

- **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 title:
    \[
    \text{COMPON=CEL,COMPID=568819801,ISSUER=CELAFRR,MODULE=CELAECT+????, ABEND=U4039,REASON=00000000}
    \]
    is taken.
  - For UAIMM, an SVC dump of the original abend/program interrupt with a title like:
    \[
    \text{COMPON=CEL,COMPID=568819801,ISSUER=CELAFRR,MODULE=CELAECT+????, ABEND=S00C9,REASON=00000009}
    \]
    (where the ABEND and REASON values are those of the original abend/program interrupt) is taken.

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

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 488.

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 143 on page 381 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 389.

The CEE3DMP was generated by the C program CELQSAMP shown in Figure 141 on page 379. CELQSAMP uses the DLL CELQDLL shown in Figure 142 on page 381.
```c
#pragma options(SERVICE("1.8"),NODOPT,GONUM)
#pragma runopts(TERMTHDACT(UADUMP),POSIX(ON))
#pragma runopts(TRACE(ON,1M,NODUMP,LE=1),HEAPCHK(ON))
#pragma runopts(RPTSTG(ON))
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <dll.h>

typedef void* FUNC(void *);

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

void *thread_func(void *parm) {
    printf(">>> Thread_func: %s locking mutex\n",parm);
    pthread_mutex_lock(&mut);
    printf(">>> Thread_func: %s exiting\n",parm);
    pthread_exit(NULL);
}

int main() {
    dllhandle * handle;
    FUNC * fp;
    FILE* fp1;
    FILE* fp2;

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

    printf("Query DLL...\n");
    fp = (FUNC *)dllqueryfn(handle,"div_zero");
    if (fp == NULL) {
        perror("Could not find thread_func");
        exit(107);
    }

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

Figure 141. The C program CELQSAMP (Part 1 of 2)
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[1], NULL, thread_func, (void *)t2) == -1) {
    perror("Could not create thread #2");
    exit(104);
}
printf("Write to some files...\n");
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...\n");
fp(NULL);
printf("Error -- Should not get here\n");
exit(110);

Figure 141. The C program CELQSAMP (Part 2 of 2)
/* DLL containing div_zero */
#pragma options(SERVICE("1.8"),NOOPT,GONUM)
#pragma export(div_zero)
#include <stdio.h>
#include <stdlib.h>

/**************************************************************************/
/* div_zero: Cause divide by zero exception */
/**************************************************************************/

void *div_zero(void *parm)
{
    int i = 0;
    printf("Divide by zero...
");
    i = 1/i;
    printf("Error -- Should not get here. i=%d\n",i);
    exit(110);
}

Figure 142. The C DLL CELQDLL

Figure 143. Example dump using CEE3DMP (Part 1 of 9)
Figure 143. Example dump using CEE3DMP (Part 2 of 9)
Figure 143. Example dump using CEE3DMP (Part 3 of 9)
CIB for div_zero(00000001082F8E00)
+0000 0000001082F800 C3C9C240 00000000 00000000 00000000 | CIB ............ |
+0010 0000001082F810 00000000 00000000 01900000 00000000
+0020 0000001082F820 00000000 00000000 00000006 59C3C5C5 ..........F.CEE
+0030 0000001082F830 00000000 00000000 00000001 082F8F90 .............
+0040 0000001082F840 00000089 59C3C5C5 00000000 00000004 00000001 .CEE
+0050 0000001082F850 00000000 00000000 00000001 082F8FB0 ..........1.
+0060 0000001082F860 00000000 25188100 00000000 00000000 00000000 .1
+0070 0000001082F870 00000001 0827F680 00000000 257585F0 ........W....0
+0080 0000001082F880 00000000 00000000 00000000 00000000 00000000 ...0
+0090 0000001082F890 00000000 00000000 00000000 00000000 00000000 .1
+00A0 0000001082F8A0 00000000 00000000 00000000 00000000 00000000 .1
+00B0 0000001082F8B0 - +0004FF 000000001082F8FF ..........same as above0
+00C0 0000001082F8C0 482240A0 00000000 940C9000 00000000 ...n
+00D0 0000001082F8D0 00000000 00000000 250000C0 ...........
+00E0 0000001082F8E0 00000001 082FF800 00000000 082FF808 ..........0.
+00F0 0000001082F8F0 00000001 082FF810 00000000 082FF818 ..........0.
+0100 0000001082F900 00000001 082FF820 00000000 082FF828 ..........0.
+0110 0000001082F910 00000000 00000000 250000C0 ...........
+0120 0000001082F920 00000001 082FF800 00000000 082FF808 ..........0.
+0130 0000001082F930 00000000 257585F0 00000000 00000000 00000000 ...
+0140 0000001082F940 00000000 00000000 00000000 00000000 00000000 ...
+0150 0000001082F950 00000001 082FF810 00000000 082FF818 ..........1
+0160 0000001082F960 00000014 00000000 00000000 00000000 00000000 ...
+0170 0000001082F970 00000000 00000000 00000000 00000000 00000000 ...
+0180 0000001082F980 00000001 00000000 00000000 00000000 00000000 ...

DSA for main: 0000001082F980
+000000 R4........ 000000001082F8F80 RS....... 00000000108300070 R6....... 00000000250000D8
+000018 R7........ 000000002500635E R8....... 000000000006F0 R9....... 00000000250000E9
+000030 R10........ 0000000025001480 R11....... 00000000108FCE5E0 R12....... 000000100005340
+000048 R13........ 00000000006F58 R14....... 0000000025006098 R15....... 0000000000001F
+000060 reserved. 00000000000000 reserved. 00000000000000 HPTRAN... 00000000000000
+000078 reserved. 00000000000000 00000000000000 00000000000000

[9] Storage for Active Routines:

DSA frame(00000001082F8E00)
+0000 0000001082F820 00000000 00000000 00000000 00000000 00000000 | .6
+0010 0000001082F830 00000000 00000000 00000000 00000000 00000000 | .6
+0020 0000001082F840 00000000 00000000 00000000 00000000 00000000 | .6
+0030 0000001082F850 00000000 00000000 00000000 00000000 00000000 | .6
+0040 0000001082F860 00000000 00000000 00000000 00000000 00000000 | .6
+0050 0000001082F870 00000000 00000000 00000000 00000000 00000000 | .6
+0060 0000001082F880 00000000 00000000 00000000 00000000 00000000 | .6
+0070 0000001082F890 00000000 00000000 00000000 00000000 00000000 | .6
+0080 0000001082F8A0 00000000 00000000 00000000 00000000 00000000 | .6
+0090 0000001082F8B0 00000000 00000000 00000000 00000000 00000000 | .6

Figure 143. Example dump using CEE3DMP (Part 4 of 9)
Figure 143. Example dump using CEE3DMP (Part 5 of 9)
Figure 143. Example dump using CEE3DMP (Part 6 of 9)
Figure 143. Example dump using CEE3DMP (Part 7 of 9)
Heap Storage Diagnostics
All storage has been freed.

Figure 143. Example dump using CEE3DMP (Part 8 of 9)
Sections of the Language Environment dump

The sections of the dump listed here appear independently of the Language Environment-conforming languages used.

[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.”

Figure 143. Example dump using CEE3DMP (Part 9 of 9)
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 userid associated to the CEEDUMP.

[2] **CEE3845I CEEDUMP Processing started.**

Message CEE3845I 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] **Enclave Information**

These sections show information that is specific to an enclave.

[3] **Enclave Identifier**
This statement names the enclave for which information in the dump is provided.


These sections show information that is specific to a thread. When multiple threads are dumped, these sections will appear for each thread.

[4] Information for thread

This section shows the system identifier for the thread. Each thread has a unique identifier.

[5] Traceback

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

- DSA number
  A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback.

- Entry
  For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, then the string "** NoName **" will appear.

- Entry point offset

- 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 359.

- 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).

- Program unit
  The primary entry point of the external procedure. For C routines, this is the compile unit name. For Language Environment-conforming assemblers, this is the ENTNAME = value on the CELQPRLG macro.

  If your routine was compiled with the compile options to generate statement numbers then the program unit name displayed under this column will appear as follows:
  - If your compiled routine is in a partitioned data set then only the member will be output.
  - If your compiled routine is in a sequential data set then only the last qualifier will be shown.
  - If your compiled routine is in an UNIX filename then only what fits of the filename will be displayed in a line.

  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.

- 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 or exception.

The second part contains:

- **DSA number**
  A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback.

- **Stack frame (DSA) address**
- **Entry point address**
- **Program unit address**
- **Program unit offset**
  The offset of the last instruction to run in the routine. If the offset is a negative number, zero, or a very large positive number, the routine associated with the offset probably did not allocate a save area or the routine could have been called using SVC-assisted linkage. Adding the program unit address to the offset gives you the location of the current instruction in the routine. This offset is from the starting address of the routine.

- **Compile Date**
- **Attributes**
  The attributes of the compile unit including whether character data is being treated as EBCDIC or ASCII and whether floating point data is being treated as IEEE or hexadecimal.

The third part, which is also referred to as ‘Fully Qualified Names’ section, contains the following:

- **DSA number**
- **Entry**
- **Program unit**
  Similar to the Program Unit column in part 1 except that the server name and the complete program unit (PU) name will be displayed. A PU name will appear here only if it was compiled using compile options to produce statement numbers.

- **Load Module**
  The complete pathname of a load module name residing in an UNIX filename will be displayed here if available. The load module's full pathname will be displayed if the PATH environment variable is set such that the pathname of the load module's directory appears before the current directory (.). For load modules found in data sets, the same output shown in the traceback part 1 will also be displayed here.

[6] **Condition Information for Active Routines**

This section 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.

[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

This 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

This section 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.

This section 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

This section lists the Language Environment run-time options in effect when the routine was executed.


This section 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.


Message CEE3846I identifies the end of the Language Environment dump processing. Similarly, message CEE3845I identifies the start of the dump processing. Message number CEE3846I can be used to locate the end of the previous CEEDUMP report when scanning backward in a data set that contains several CEEDUMP reports.

Generating a system dump

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

DYNDUMP(hlq,DYNAMIC,TDUMP)

You can use the DYNDUMP 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 375.

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:

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 375.

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.
When you are done, you have a generated system dump in a batch run-time environment.

**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 command:
       ```sh
       export _BPXK_MDUMP=filename
       ```
       where *filename* is a fully qualified data set name with DCB information: LRECL=4160, BLKSIZE=4160, and RECFM=FBS.
       **Example:**
       ```sh
       export _BPXK_MDUMP=hlq.mydump
       ```
     - To write the system dump to an HFS file, issue the command:
       ```sh
       export _BPXK_MDUMP=filename
       ```
       where *filename* is a fully qualified HFS filename.
       **Example:**
       ```sh
       export _BPXK_MDUMP=/tmp/mydump.dmp
       ```
  2. Specify Language Environment run-time options:
     ```sh
     export _CEE_RUNOPTS="termthdact(suboption)"
     ```
     where *suboption* = UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details regarding the TERMTHDACT suboptions, see "Generating a Language Environment dump with TERMTHDACT" on page 375.
  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](https://www.ibm.com/support/docview/groups.htm?context=en&rs=5713&uid=swg21412632).

- **Using DYNDUMP**
  1. Specify Language Environment run-time options:
     ```sh
     export _CEE_RUNOPTS="termthdact(suboption),DYNDUMP(hlq,DYNAMIC,TDUMP)"
     ```
     where:
     - *suboption* = UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details regarding the TERMTHDACT suboptions, see "Generating a Language Environment dump with TERMTHDACT" on page 375.
     - *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](https://www.ibm.com/support/docview/groups.htm?context=en&rs=5713&uid=swg21412632).
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

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

- Ensure that your IPCS job can find the CEEIPCSP member.
  - IPCS provides an exit control table with imbed statements to enable other products to supply exit control information. The IPCS default table, BLSCECT, normally in the SYS1.PARMLIB library, has the following entry for Language Environment:
    
    ```
    IMBED MEMBER(CEEIPCSP) ENVIRONMENT(IPCS)
    ```
  - The Language Environment-supplied CEEIPCSP member, installed in the SYS1.PARMLIB library, contains the Language Environment-specific entries for the IPCS exit control table.
  - Provide an IPCSPARM DD statement to specify the libraries containing the IPCS control tables.

  Example:

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

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

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

  ```
  EXIT EP(CEEEANLZ) ANALYZE
  ```

Language Environment IPCS Verbexit – LEDATA

Use the LEDATA Verbexit 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

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

- Report Type Parameters:
  - [ AUTH ]
  - [ NTHREADS(value) ]
  - [ SUM ]
  - [ HEAP | STACK | SM ]
  - [ HPT(number) [ HPTTCB (address) ] [ HPTCELL(address) ] [ HPTLOC(location) ] ]
  - [ CM ]
  - [ MH ]
  - [ CEEDUMP ]
  - [ COMP((value)) ]
  - [ PTBL(value) ]
  - [ ALL ]

- Data Selection Parameters:
  - [ DETAIL | EXCEPTION ]

- Control Block Selection Parameters:
  - [ CAA(caa-address) ]
  - [ DSA(dsa-address) ]
  - [ TCB(tcb-address) ]
  - [ ASID(address-space-id) ]
  - [ LAA(laa-address) ]

Parameters

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)  [ 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(value)
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. User 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.

ALL
Requests all above reports, as well as C/C++ reports.

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.
- 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, 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.

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.
**Note:** 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 418.

**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.

**Note:** 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.

**Usage Note:** Only hexadecimal characters can be specified as addresses provided in LEDATA parameters. Special characters cause the formatter to fail. Therefore, in order to specify a 64 bit address as a parameter, it must be in the form like 123456789 instead of 1_23456789.

**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 144 on page 403 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 141 on page 379." Sections of the Language
Environment LEDATA Verbexit formatted output on page 412 describes the information contained in the formatted output. Ellipses are used to summarize some sections of the dump.

For easy reference, the sections of the following dump are numbered to correspond with the descriptions in "Sections of the Language Environment LEDATA Verbexit formatted output" on page 412.
ALL

64 BIT LANGUAGE ENVIRONMENT DATA

Language Environment Product 04 V01 R09.00

[1] Information for enclave main

[2] Information for thread 253E019000000000
   TCB Address: 007F050
   CAA Address: 0000001_000057B1
   PCB Address: 0000001_00003C0

[3] Registers and PSW:
   GPR0.... 00000000B4000000  GPR1.... 00000000B4000000  ...
   GPR6.... 00000000253C459A  GPR7.... 00000000253C459A  ...
   GPR8.... 00000000253C459A  GPR9.... 00000000253C459A  ...
   GPR10.... 00000000253C459A  GPR11.... 00000000253C459A  ...
   GPR12.... 00000000253C459A  GPR13.... 00000000253C459A  ...
   GPR14.... 00000000253C459A  GPR15.... 00000000253C459A  ...
   PSW.... 07851401 00000000 00000000 253C459A

[4] Traceback:

   DSA  Entry  E Offset  Statement  Load Mod  Program Unit  Service Status
   1  CEEHDMP  +0000000A  CELQLIB  CEEHSMP  0190  Call
   2  CEEHDSP  +00003A88  CELQLIB  CEEHSMP  0190  Call
   3  CEEOS16J  +00009A4E  CELQLIB  CEEOS16J  0190  Call
   4  CELQHROD  +0000024E  CELQLIB  CELQHROD  0190  Call
   5  CELQHROD  +00000000  CELQLIB  CELQHROD  0190  Call
   6  CELQHROD  +0000024E  CELQLIB  CELQHROD  0190  Call
   7  div_zero  +00000040  CELQLIB  CELQHROD  1.4  Exception
   8  main  +00000468  CELQHROD  CELQHROD  1.2  Call
   9  CELQINIT  +0000134C  CELQLIB  CELQINIT  0190  Call

   DSA  DSA Addr  E Addr  PU Addr  PU Offset  Comp Date  Compile Attributes
   1  00000001_082F9A00 00000000_253C4500 00000000 00000000 00000000 00000000 00000000
   2  00000001_082F9AC0 00000000_253C4500 00000000 00000000 00000000 00000000 00000000
   3  00000001_082FD3E0 00000000_253C4500 00000000 00000000 00000000 00000000 00000000
   4  00000001_082FD6E0 00000000_253C4500 00000000 00000000 00000000 00000000 00000000
   5  00000001_082FE020 00000000_253C4500 00000000 00000000 00000000 00000000 00000000
   6  00000001_082FEE40 00000000_253C4500 00000000 00000000 00000000 00000000 00000000
   7  00000001_082FF000 00000000_253C4500 00000000 00000000 00000000 00000000 00000000
   8  00000001_082FF180 00000000_253C4500 00000000 00000000 00000000 00000000 00000000
   9  00000001_082FF290 00000000_253C4500 00000000 00000000 00000000 00000000 00000000

   Control Blocks Associated with the Thread:
   Thread Synchronization Queue Element (SQEL): 00000000_257520A0
   same as above

   Enclave Control Blocks:
   Mutex and Condition Variable Blocks (MCV8+MHT+CHT): 00000001_089100B8
   same as above

Figure 144. Example of formatted output from LEDATA Verexit (Part 1 of 10)
Figure 144. Example of formatted output from LEDATA Verbexit (Part 2 of 10)
Figure 144. Example of formatted output from LEDATA Verbexit (Part 3 of 10)
Figure 144. Example of formatted output from LEDATA Verexit (Part 4 of 10)
Figure 144. Example of formatted output from LEDATA Verbexit (Part 5 of 10)
Figure 144. Example of formatted output from LEDATA Verbx (Part 6 of 10)
Figure 144. Example of formatted output from LEDATA Verbexit (Part 7 of 10)
Figure 144. Example of formatted output from LEDATA Verbexit (Part 8 of 10)
Figure 144. Example of formatted output from LEDATA Verbind (Part 9 of 10)
<table>
<thead>
<tr>
<th>Language Environment LEDATA Verbexit formatted output</th>
</tr>
</thead>
<tbody>
<tr>
<td>The sections of the output listed here appear independently of the Language Environment-conforming languages used.</td>
</tr>
</tbody>
</table>

Figure 144. Example of formatted output from LEDATA Verbexit (Part 10 of 10)
CEEDUMP Formatted Control Blocks

These sections are included when the CEEDUMP parameter is specified on the LEDATA invocation.

NTHREADS data

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 398.

Enclave Identifier

This statement names the enclave for which information is provided.

Information for thread

This section shows the system identifier for the thread. Each thread has a unique identifier.

Registers and PSW

This section 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.

Traceback

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

- 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.

- Entry
  For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, then the string "** NoName **" will appear.

- Entry point offset

- Load module

- 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.

- 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:
- **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.
- **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.

[5] **Control Blocks Associated with the Thread**

This section lists the contents of the thread synchronization queue element (SQEL).

[6] **Enclave Control Blocks**

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.

[7] **Language Environment Trace Table**

If the TRACE run-time option was set to ON, this section shows the contents of the Language Environment trace table.

[8] **Process Control Blocks**

If the POSIX run-time option was set to ON, this section lists the contents of the process level latch table.

[9] - [17] **Summary**

These sections are included when the SUMMARY parameter is specified on the LEDATA invocation.

[9] **Summary Header**

The summary header section contains:
- Address of Thread control block (TCB)
- Release number
- Address Space ID (ASID)

[10] **Active Members List**

This list of active members is extracted from the enclave member list (MEML).

This section formats the contents of the Language Environment library anchor area (LAA). See z/OS Language Environment Vendor Interfaces for a description of the fields in the LAA.

[12] CEELCA

This section 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

This section 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

This section formats the contents of the Language Environment process control block (PCB), and the process level member list.

[15] CEERCB

This section formats the contents of the Language Environment region control block (RCB).

[16] CEEEDB

This section formats the contents of the Language Environment enclave data block (EDB), and the enclave level member list.

[17] Run-Time Options

This section 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.

This section 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 418.

[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.

This section 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.

This section formats information related to preinitialization. See section PTBL LEDATA output 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 145 on page 417 illustrates the output produced when the Verbexit LEDATA command is invoked with the PTBL parameter.
Figure 145. 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 146 on page 419 illustrates the output produced by
specifying the HEAP option. "Heap report sections of the LEDATA output" on page
427 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. Ellipses are used to summarize some
sections of the dump.

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

Figure 145. Example of formatted PTBL output from LEDATA Verbexit (Part 2 of 2)
### Heap Storage Control Blocks

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

<table>
<thead>
<tr>
<th>ENSQ:</th>
<th>00000001_00100108</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>EYE_CATCHER:ENSQ</td>
</tr>
<tr>
<td>+000008</td>
<td>HEAPFREE_VAL:00000000</td>
</tr>
<tr>
<td>+000014</td>
<td>IPT_TOKEN:0000000000000000</td>
</tr>
<tr>
<td>+000024</td>
<td>HEAPLOCKWORD:00000000</td>
</tr>
<tr>
<td>+000030</td>
<td>UHEAP64:CB7C3DB</td>
</tr>
<tr>
<td>+000060</td>
<td>LHEAP64:CB7C3DB</td>
</tr>
<tr>
<td>+000090</td>
<td>UHEAP31:CB7C3DB</td>
</tr>
<tr>
<td>+0000C0</td>
<td>LHEAP31:CB7C3DB</td>
</tr>
<tr>
<td>+0000F0</td>
<td>UHEAP24:CB7C3DB</td>
</tr>
<tr>
<td>+000120</td>
<td>LHEAP24:CB7C3DB</td>
</tr>
<tr>
<td>+000174</td>
<td>IPT_TCB:007FF050</td>
</tr>
<tr>
<td>+000188</td>
<td>STSB:00000000_00000000</td>
</tr>
<tr>
<td>+000198</td>
<td>TOKEN:7F7547D8</td>
</tr>
<tr>
<td>+0001A0</td>
<td>THDLHEAP64:CB7C3DB</td>
</tr>
<tr>
<td>+0001D0</td>
<td>IOHEAP64:CB7C3DB</td>
</tr>
<tr>
<td>+000200</td>
<td>IOHEAP31:CB7C3DB</td>
</tr>
<tr>
<td>+000230</td>
<td>IOHEAP24:CB7C3DB</td>
</tr>
<tr>
<td>+000260</td>
<td>SM_CELL_BLOCK:00000000_00000000</td>
</tr>
</tbody>
</table>

### User Heap64 Control Blocks

<table>
<thead>
<tr>
<th>HPCQ:</th>
<th>00000001_00100138</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>EYE_CATCHER:HPCQ</td>
</tr>
<tr>
<td>+000018</td>
<td>INITSIZE:00000000</td>
</tr>
<tr>
<td>+00002C</td>
<td>OPTIONS:80000000</td>
</tr>
<tr>
<td>HPSQ:</td>
<td>00000001_00005058</td>
</tr>
<tr>
<td>+000000</td>
<td>BYTES_ALLOC:00000000</td>
</tr>
<tr>
<td>+000008</td>
<td>CURR_ALLOC:00000000</td>
</tr>
<tr>
<td>+000018</td>
<td>FREE_REQ:00000000</td>
</tr>
<tr>
<td>+000028</td>
<td>FREEMAINS:00000000</td>
</tr>
<tr>
<td>THNQ:</td>
<td>00000001_001005B0</td>
</tr>
<tr>
<td>+000000</td>
<td>EYE_CATCHER:THNQ</td>
</tr>
<tr>
<td>+000010</td>
<td>PREV:00000001_00100138</td>
</tr>
<tr>
<td>+000020</td>
<td>SEGMENT:00000001_08300000</td>
</tr>
<tr>
<td>HANQ:</td>
<td>00000001_08300000</td>
</tr>
<tr>
<td>+000000</td>
<td>EYE_CATCHER:HANQ</td>
</tr>
<tr>
<td>+000020</td>
<td>SEGMENT:00000001_08300000</td>
</tr>
<tr>
<td>+000030</td>
<td>SEG_LEN:00000000</td>
</tr>
</tbody>
</table>

---

**Figure 146. Example formatted detailed heap segment report from LEDATA Verbexit (Part 1 of 9)**
Figure 146. Example formatted detailed heap segment report from LEDATA Verexit (Part 2 of 9)
Chapter 12. Using Language Environment AMODE 64 debugging facilities 421

Figure 146. Example formatted detailed heap segment report from LEDATA Verbexit (Part 3 of 9)
[1] Free Storage Tree for Heap Segment 0000001

<table>
<thead>
<tr>
<th>Depth</th>
<th>Node Address</th>
<th>Node Length</th>
<th>Parent Node Address</th>
<th>Left Node Address</th>
<th>Right Node Address</th>
<th>Left Length</th>
<th>Right Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000001_08E00000</td>
<td>0000000000113E0</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>00000001_08E00000</td>
<td>0000000000113E0</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>


To display entire segment: IP LIST 00000001_08E00000 LEN(X'0000000000020000') ASID(X'002B')

00000001_08E00000: Allocated storage element, length=0000000000020000.

To display: IP LIST 00000001_08E00000 LEN(X'0000000000010000') ASID(X'002B')

00000001_08E00000: Allocated storage element, length=0000000000010000.

Summary of analysis for Heap Segment 0000001:

Amounts of identified storage:
- Allocated: 00348300
- Total: 003FFFC0

00000001_08E00000 bytes of storage were not accounted for.

No errors were found while processing this heap segment.

THQ:
- 00000001_00100670
- 00000001_00100640

MAP:
- 00000001_00100640
- 00000001_00100670

HANQ:
- 00000001_08E00000
- 00000001_08E00000

THQ:
- 00000001_00100670
- 00000001_00100640

This is the last heap segment in the current heap.


To display entire segment: IP LIST 00000001_08FEEC20 LEN(X'0000000000020000') ASID(X'002B')

00000001_08FEEC20: Allocated storage element, length=0000000000020000.

To display: IP LIST 00000001_08FEEC20 LEN(X'0000000000010000') ASID(X'002B')

00000001_08FEEC20: Allocated storage element, length=0000000000010000.

Summary of analysis for Heap Segment 00000001:

Amounts of identified storage:
- Allocated: 003FFFC0
- Total: 003FFFC0

00000001_08FEEC20 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 146. Example formatted detailed heap segment report from LEDATA Verbexit (Part 4 of 9)
Figure 146. Example formatted detailed heap segment report from LEDATA Verbexit (Part 5 of 9)
Figure 146. Example formatted detailed heap segment report from LEDATA Verbexit (Part 6 of 9)
Figure 146. Example formatted detailed heap segment report from LEDATA Verbexit (Part 7 of 9)
Figure 146. Example formatted detailed heap segment report from LEDATA Verbexit (Part 8 of 9)
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.


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 zero
- 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


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  

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  
• 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 147 on page 430 illustrates the details of heappool report. "Heappool report sections of the LEDATA output" on page 435 describes the information contained in the formatted output.
Figure 147. Example formatted detailed heap pool report from LEDATA Verbexit (Part 1 of 5)
Figure 147. Example formatted detailed heapool report from LEDATA Verbexit (Part 2 of 5)
Figure 147. Example formatted detailed heap pool report from LEDATA Verbexit (Part 3 of 5)
Figure 147. Example formatted detailed heapool report from LEDATA Verbexit (Part 4 of 5)
There are no extents for this pool.

There are no extents for this pool.

There are no extents for this pool.

There are no extents for this pool.
Heappool report sections of the LEDATA output
The Heappool report provides the information regarding 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.

[1]Free Chain Validation
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.

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 in order to identify the reason for the storage allocation.

The formatter validates if cell pool number in header is correct.

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. The argument value is the id of the pool to be formatted in the report.
Figure 148. Example formatted detailed HEAPPOOLS trace report from LEDATA Verbexit (Part 1 of 5)
Figure 148. Example formatted detailed HEAPPOOLS trace report from LEDATA Verbexit (Part 2 of 5)
HEAPPOOLS Trace Table

POOLID: 3 ASID: 001F AVAILABLE ENTRIES: 8 OF 8

Figure 148. Example formatted detailed HEAPPOOLS trace report from LEDATA Verbexit (Part 3 of 5)
Figure 148. Example formatted detailed HEAPPOOLS trace report from LEDATA Verbexit (Part 4 of 5)
### Trace Header

HEAPPOOLS trace header information.

### Pool Information

Information includes the number of the pool (POOLID) which 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.

### Timestamp

Figure 148. Example formatted detailed HEAPPOOLS trace report from LEDATA Verbexit (Part 5 of 5)
The time this trace entry was taken.

**Note:** The trace entries are formatted in reverse order (most recent trace entry first).

[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 ALL parameter is specified and C/C++ is active in the dump. Figure 149 on page 443 illustrates the C/C++-specific output produced. The system dump being formatted was obtained by specifying the TERMTTHDACT(UADUMP) run-time option when running the program CELQSAMP. Figure 5 on page 44 describes the information contained in the formatted output. Ellipses are used to summarize some sections of the dump.

For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 1 of 12)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 2 of 2)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 3 of 12)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 4 of 12)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 5 of 12)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 6 of 12)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 7 of 12)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 8 of 12)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 9 of 12)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 10 of 12)
Figure 149. Example formatted C/C++ output from LEDATA Verbexit (Part 11 of 12)
C/C++-specific sections of the LEDATA output

For the LEDATA output:

[1] CGEN

This section formats the C/C++-specific portion of the Language Environment common anchor area (CAA).

[2] CGENE

This section formats the extension to the C/C++-specific portion of the Language Environment common anchor area (CAA).

[3] CEDB

This section formats the C/C++-specific portion of the Language Environment enclave data block (EDB).

[4] CTHD
This section formats the C/C++ thread-level control block (CTHD).

[5] CPCB

This section formats the C/C++-specific portion of the Language Environment process control block (PCB).

[6] CIO

This section formats the C/C++ IO control block (CIO).

[7] File Control Blocks

This section formats the C/C++ file control block (FCB). The FCB and its related control blocks represent the information needed by each open stream.

Related Control Blocks

**FFIL**  This section formats the header of the C/C++ file control block (FCB).

**FSCE**  The file specific category extension control block. The FSCE represents the specific type of IO being performed. The following is a list of FSCEs that may be formatted.

- HFSF — UNIX file system file
- HSPF — Hiper-Space file
- INTC — Intercept file
- MEMF — Memory file
- OSNS — OS no seek
- OSFS — OS fixed text
- OSVF — OS variable text
- OSUT — OS undefined format text
- TDQF — CICS Transient Data Queue file
- TERM — Terminal file
- VSAM — VSAM file

Other FSCEs will be displayed using a generic overlay.

**OSIO**  The OS IO interface control block.

**OSIOE**  The OS IO extended interface control block.

**DCB**  The data control block. For more information about the DCB, see [z/OS DFSMS Macro Instructions for Data Sets](https://www.ibm.com/docs/en/zos/2.4.0?topic=dfsms-context-macros).

**DCBE**  The data control block extension. For more information about the DCBE, see [z/OS DFSMS Macro Instructions for Data Sets](https://www.ibm.com/docs/en/zos/2.4.0?topic=dfsms-context-macros).


**JFCBX**  The job file control block extension (JFCBX).
Mbuf
The message buffer control block (MBUF).

[8] Memory File Control Blocks
This section formats the C/C++ memory file control block (MFCB).

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 150 on page 457 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.
### Authorized Language Environment Control Blocks

**Authorized Language Environment Control Blocks**

1. **ALEC**: 00000000_7F6F7000
   - Flags1: 40000000
   - UseCount: 00000000
   - ASATable@1: 7F6E8414
   - ASATable@2: 7F6E8754
   - ASATable@3: 00000000
   - ASATable@4: 00000000
   - MCallRtn: 82991A80
   - UCallRtn: 82999DB8
   - LatchSetTok: 7F6C0B40
   - StackCPID: 7F6C3F00
   - AROTCB: 008FF028
   - EnvTypeNum: 00000000
   - WorkECB: 808E6F10
   - AROTK Token: 0000006C
   - ALELVT: 00000000
   - SLELVT: 00100140
   - FuncTable@: 00000000
   - WorkQueue: 00000000
   - ALESCount: 00000000
   - SystemRtnCode: 00000000
   - SystemRsnCode: 00000000
   - SystemRtnCodeJr: 00000000
   - WorkerTCB: 008D8E88
   - SystemOCB@: 00000000

2. **Load Module Control Blocks**

   **Queue #0**
   - ALMI: 00000000_01000F40
     - Flags1: 40000000
     - UseCount: 00000000
     - LoadPoint: 00000000
     - EntryPoint: 00000000
   - ALMI: 00000000_0100B40
     - Flags1: 40000000
     - UseCount: 00000000
     - LoadPoint: 00000000
     - EntryPoint: 00000000
   - ALMI: 00000000_0100540
     - Flags1: 40000000
     - UseCount: 00000000
     - LoadPoint: 00000000
     - EntryPoint: 00000000
   - ALMI: 00000000_0100740
     - Flags1: 40000000
     - UseCount: 00000000
     - LoadPoint: 00000000
     - EntryPoint: 00000000
   - ALMI: 00000000_0100340
     - Flags1: 40000000
     - UseCount: 00000000
     - LoadPoint: 00000000
     - EntryPoint: 00000000

3. **User Managed Control Blocks**

   **Queue #1**
   - ALMI: 00000000_01100740
     - Flags1: 40000000
     - UseCount: 00000000
     - LoadPoint: 00000000
     - EntryPoint: 00000000

---

**Figure 150. Example of formatted AUTH output from LEDATA Verbexit (Part 1 of 4)**
Figure 150. Example of formatted AUTH output from LEDATA Verbexit (Part 2 of 4)
Figure 150. Example of formatted AUTH output from LEDATA Verbexit (Part 3 of 4)
Figure 150. Example of formatted AUTH output from LEDATA Verbexit (Part 4 of 4)
Sections of the AUTH LEDATA Verbexit formatted output

[1]ALEC

The ALEC is the 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).

[2]Load Module Control Blocks

This section is the formatted representation of a table of ALMI control blocks. Each ALMI represents a module that was loaded by Preinitialized Environments for Authorized Programs.

[3]User Managed Control Blocks

This section contains control blocks for all user managed environments. A user managed environment is initialized when the CELAAUTH macro is invoked with REQUEST=USERINIT.

[4]-[5]Control Blocks for one user managed environment

These sections are repeated for each user managed environment that was initialized.

[4]ALEI

Each ALEI control block represents one environment.

[5]Routine Control Blocks

This section is the 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.

[6]System Managed Control Blocks

This section contains control blocks for all system managed environments. A set of system managed environments is initialized when the CELAAUTH macro is invoked with REQUEST=MNGDINIT.

[7]-[11]Control Blocks for one set of system managed environments that was initialized

These sections are repeated for each set of system managed environments that was initialized.

[7]ALES

Each ALES represents a set of system managed environments.

[8]-[11]Control blocks for one environment definition entry
These sections are repeated for every environment definition entry (AEDE) that was specified when the set of system managed environments was initialized.

[8] ETINDEX and ALESETE

The ETINDEX is the environment definition entry index value and the ALESETE represents the environment definition entry.

[9] Routine Control Blocks

This section is the 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.

[10]-[11] Control blocks for one system managed environment

These sections are repeated for every environment associated with the ETINDEX and ALESETE above.

[10] ALEI

Each ALEI control block represents one environment. The ALEIs in this section represent system managed environments.

[10] ALRI

This section contains the ALRI control blocks for each routine that was called in the environment identified by the ALEI above. This section does not appear if the environment has not been used to call a routine.

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”.

Syntax

```
CBF ADDRESS STRUCTURE (cbname)
```

<table>
<thead>
<tr>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>The address of the control block in the dump. This is determined by browsing the dump or running the LEDATA Verbexit.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cbname</th>
</tr>
</thead>
</table>
| The name of the control block to be formatted. The control blocks that can be individually formatted are listed in Table 28 on page 463. 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 in order to uniquely define the Language Environment control block names to IPCS.

For an example of the display which is the result of the command, see Figure 151:

CBF 15890 struct(CECAA)

Table 28. Language Environment control blocks which can be individually formatted

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELCIB</td>
<td>Condition Information Block</td>
</tr>
<tr>
<td>CELCIBH</td>
<td>Condition Information Block Header</td>
</tr>
<tr>
<td>CELDLLF</td>
<td>DLL Failure Control Block</td>
</tr>
<tr>
<td>CELDSA</td>
<td>Dynamic Storage Area</td>
</tr>
<tr>
<td>CELDSATR</td>
<td>XPLINK Transition Area</td>
</tr>
<tr>
<td>CELEDIB</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>CELLLAA</td>
<td>Library Anchor Area</td>
</tr>
<tr>
<td>CELLLCA</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>
</tbody>
</table>
Table 28. Language Environment control blocks which can be individually formatted (continued)

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 29. Preinitialized Environments for Authorized Programs control blocks which 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 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. Under abnormal termination, the following dump contents are generated:

- TERMTHDACT(QUIET) generates a Language Environment dump containing the trace table only
- TERMTHDACT(MSG) generates a Language Environment dump containing the trace table only
- TERMTHDACT(TRACE) generates a Language Environment dump containing the trace table and the traceback
- TERMTHDACT(DUMP) generates a Language Environment dump containing thread/enclave/process storage and control blocks (the trace table is included as an enclave control block)
- TERMTHDACT(UAONLY) generates a system dump of the user address space
TERMTHDACT(UATRACE) generates a Language Environment dump that contains traceback information, and a system dump of the user address space.

TERMTHDACT(UADUMP) generates a 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.

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

Note: TRAP(ON,NOSPIE) must be in effect. When TRAP(ON,SPIE) is in effect, UAIMM equals UAONLY results. For software raised conditions or signals, UAIMM is the same as UAONLY.

Under normal termination, with the TRACE 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 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 152. Format of the trace table entry

Following is a definition of each field:

**Time**
The 64-bit value obtained from a store clock (STCK).

**Thread ID**
The 8-byte thread ID of the thread that is adding the trace table entry.

**Member ID and Flags**
Contains 2 fields:

- **Member ID**
The 1-byte member ID of the member making the trace table entry, as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>CEL</td>
</tr>
<tr>
<td>03</td>
<td>C/C++</td>
</tr>
<tr>
<td>08</td>
<td>DCE</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>
</tbody>
</table>

**When LE=1 is specified:** The following C/C++ records may be generated.

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>

For a detailed description of these records, see "C/C++ contents of the Language Environment trace tables" on page 498.

**When LE=2 is specified:** The following Language Environment records may be generated.

<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>
</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>000000304</td>
<td>MX</td>
<td>AW</td>
<td>Mutex acquire after wait</td>
</tr>
<tr>
<td>01</td>
<td>000000305</td>
<td>MX</td>
<td>B</td>
<td>Mutex busy (Trylock failed)</td>
</tr>
<tr>
<td>01</td>
<td>000000306</td>
<td>MX</td>
<td>I</td>
<td>Mutex increment (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>000000307</td>
<td>MX</td>
<td>D</td>
<td>Mutex decrement (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>000000315</td>
<td>MX</td>
<td>IN</td>
<td>Mutex initialize</td>
</tr>
<tr>
<td>01</td>
<td>000000316</td>
<td>MX</td>
<td>DS</td>
<td>Mutex destroy</td>
</tr>
<tr>
<td>01</td>
<td>00000031D</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_SM(DESTROY) error return</td>
</tr>
<tr>
<td>01</td>
<td>000004CF</td>
<td>ME</td>
<td>FBU</td>
<td>Error - BPX_SM(fail) returns EBUSY</td>
</tr>
<tr>
<td>01</td>
<td>000004D0</td>
<td>ME</td>
<td>FIV</td>
<td>Error - BPX_SM(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>
</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>000004EB</td>
<td>ME</td>
<td>EDD</td>
<td>Mutex destroy failed (damage)</td>
</tr>
<tr>
<td>01</td>
<td>000004EC</td>
<td>ME</td>
<td>EDB</td>
<td>Mutex destroy failed (busy)</td>
</tr>
<tr>
<td>01</td>
<td>000004ED</td>
<td>ME</td>
<td>EIA</td>
<td>Mutex initialize failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000004EE</td>
<td>ME</td>
<td>EIS</td>
<td>Mutex initialize failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000004EF</td>
<td>ME</td>
<td>EF</td>
<td>Mutex release (forced by quiesce)</td>
</tr>
<tr>
<td>01</td>
<td>000004F0</td>
<td>ME</td>
<td>EP</td>
<td>Mutex program check</td>
</tr>
<tr>
<td>01</td>
<td>000004FA</td>
<td>ME</td>
<td>EDU</td>
<td>Mutex destroy failed (uninitialized)</td>
</tr>
<tr>
<td>01</td>
<td>000004FB</td>
<td>ME</td>
<td>EUI</td>
<td>Mutex uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000004FC</td>
<td>ME</td>
<td>EUO</td>
<td>Mutex unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000004FD</td>
<td>E</td>
<td>EO</td>
<td>Mutex already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000004FE</td>
<td>ME</td>
<td>EIN</td>
<td>Mutex initialization failed (duplicate)</td>
</tr>
<tr>
<td>01</td>
<td>00000508</td>
<td>CV</td>
<td>MR</td>
<td>CV release mutex</td>
</tr>
<tr>
<td>01</td>
<td>00000509</td>
<td>CV</td>
<td>MA</td>
<td>CV reacquire mutex</td>
</tr>
<tr>
<td>01</td>
<td>0000050A</td>
<td>CV</td>
<td>MW</td>
<td>CV mutex wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050B</td>
<td>CV</td>
<td>MAW</td>
<td>CV reacquire mutex after wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050C</td>
<td>CV</td>
<td>CW</td>
<td>CV condition wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050D</td>
<td>CV</td>
<td>CTW</td>
<td>CV condition timeout</td>
</tr>
<tr>
<td>01</td>
<td>0000050E</td>
<td>CV</td>
<td>CWP</td>
<td>CV wait posted</td>
</tr>
<tr>
<td>01</td>
<td>0000050F</td>
<td>CV</td>
<td>CWI</td>
<td>CV wait interrupted</td>
</tr>
<tr>
<td>01</td>
<td>00000510</td>
<td>CV</td>
<td>CTO</td>
<td>CV wait timeout</td>
</tr>
<tr>
<td>01</td>
<td>00000511</td>
<td>CV</td>
<td>CSS</td>
<td>CV condition signal success</td>
</tr>
<tr>
<td>01</td>
<td>00000512</td>
<td>CV</td>
<td>CSM</td>
<td>CV condition signal miss</td>
</tr>
<tr>
<td>01</td>
<td>00000513</td>
<td>CV</td>
<td>CBS</td>
<td>CV condition broadcast success</td>
</tr>
<tr>
<td>01</td>
<td>00000514</td>
<td>CV</td>
<td>CBM</td>
<td>CV condition broadcast miss</td>
</tr>
<tr>
<td>01</td>
<td>00000515</td>
<td>CV</td>
<td>IN</td>
<td>CV initialize</td>
</tr>
<tr>
<td>01</td>
<td>00000516</td>
<td>CV</td>
<td>DS</td>
<td>CV destroy</td>
</tr>
<tr>
<td>01</td>
<td>00000522</td>
<td>CV</td>
<td>CIN</td>
<td>Call to SMC_INIT</td>
</tr>
<tr>
<td>01</td>
<td>00000523</td>
<td>CV</td>
<td>CSD</td>
<td>Call to SMC_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>00000529</td>
<td>CV</td>
<td>CSP</td>
<td>Call to SMC_POST</td>
</tr>
<tr>
<td>01</td>
<td>0000052A</td>
<td>CV</td>
<td>CSB</td>
<td>Call to SMC_POSTALL</td>
</tr>
<tr>
<td>01</td>
<td>0000052B</td>
<td>CV</td>
<td>CSW</td>
<td>Call to SMC_WAIT</td>
</tr>
<tr>
<td>01</td>
<td>0000052C</td>
<td>CV</td>
<td>DBM</td>
<td>Shared condition broadcast - miss</td>
</tr>
<tr>
<td>01</td>
<td>0000052D</td>
<td>CV</td>
<td>DBS</td>
<td>Shared condition broadcast - success</td>
</tr>
<tr>
<td>01</td>
<td>0000052E</td>
<td>CV</td>
<td>DDS</td>
<td>Destroy (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>0000052F</td>
<td>CV</td>
<td>DIN</td>
<td>Initialize (shared mutex/CV)</td>
</tr>
<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>
</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>00000534</td>
<td>CV</td>
<td>DWP</td>
<td>Wait posted (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006CB</td>
<td>CE</td>
<td>FBT</td>
<td>Error - Invalid system TOD (shared)</td>
</tr>
<tr>
<td>01</td>
<td>000006D1</td>
<td>CE</td>
<td>FRM</td>
<td>Error - Recursive mutex (shared)</td>
</tr>
<tr>
<td>01</td>
<td>000006D2</td>
<td>CE</td>
<td>FUO</td>
<td>Error - Shared mutex unowned</td>
</tr>
<tr>
<td>01</td>
<td>000006D3</td>
<td>CE</td>
<td>FDB</td>
<td>Error - Destroy failed (busy) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D4</td>
<td>CE</td>
<td>FDU</td>
<td>Error - Destroy failed (uninitialized) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D5</td>
<td>CE</td>
<td>FP</td>
<td>Error - Program check (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D6</td>
<td>CE</td>
<td>FUI</td>
<td>Error - Shared mutex or CV uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000006D7</td>
<td>CE</td>
<td>ENV</td>
<td>Error - BPX1SMC(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000006D8</td>
<td>CE</td>
<td>EPE</td>
<td>Error - BPX1SMC(fail) returns EPERM</td>
</tr>
<tr>
<td>01</td>
<td>000006D9</td>
<td>CE</td>
<td>EAN</td>
<td>Error - BPX1SMC(fail) returns EAGAIN</td>
</tr>
<tr>
<td>01</td>
<td>000006DA</td>
<td>CE</td>
<td>EIB</td>
<td>Error - BPX1SMC failed (EBUSY)</td>
</tr>
<tr>
<td>01</td>
<td>000006DB</td>
<td>CE</td>
<td>ESC</td>
<td>Error - BPX1SMC failed</td>
</tr>
<tr>
<td>01</td>
<td>000006EB</td>
<td>CE</td>
<td>EDD</td>
<td>CV destroy failed (damage)</td>
</tr>
<tr>
<td>01</td>
<td>000006EC</td>
<td>CE</td>
<td>EDB</td>
<td>CV destroy failed (busy)</td>
</tr>
<tr>
<td>01</td>
<td>000006ED</td>
<td>CE</td>
<td>EIA</td>
<td>CV initialization failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000006EE</td>
<td>CE</td>
<td>EIS</td>
<td>CV initialization failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000006EF</td>
<td>CE</td>
<td>EF</td>
<td>CV forced by quiesce</td>
</tr>
<tr>
<td>01</td>
<td>000006F0</td>
<td>CE</td>
<td>EP</td>
<td>CV program check</td>
</tr>
<tr>
<td>01</td>
<td>000006F1</td>
<td>CE</td>
<td>EBT</td>
<td>CV invalid system TOD</td>
</tr>
<tr>
<td>01</td>
<td>000006F2</td>
<td>CE</td>
<td>EBN</td>
<td>CV invalid timespec (nanoseconds)</td>
</tr>
<tr>
<td>01</td>
<td>000006F3</td>
<td>CE</td>
<td>EBS</td>
<td>CV invalid timespec (seconds)</td>
</tr>
<tr>
<td>01</td>
<td>000006F4</td>
<td>CE</td>
<td>EPO</td>
<td>CV condition post callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F5</td>
<td>CE</td>
<td>ETW</td>
<td>CV condition timed wait callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F6</td>
<td>CE</td>
<td>EWA</td>
<td>CV condition wait callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F7</td>
<td>CE</td>
<td>ESE</td>
<td>CV condition setup callable service fail</td>
</tr>
<tr>
<td>01</td>
<td>000006F8</td>
<td>CE</td>
<td>ERM</td>
<td>CV recursive mutex</td>
</tr>
<tr>
<td>01</td>
<td>000006F9</td>
<td>CE</td>
<td>EWM</td>
<td>CV wrong mutex</td>
</tr>
<tr>
<td>01</td>
<td>000006FA</td>
<td>CE</td>
<td>EDU</td>
<td>CV destroy failed (uninitialized)</td>
</tr>
<tr>
<td>01</td>
<td>000006FB</td>
<td>CE</td>
<td>EUI</td>
<td>CV mutex or CV uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000006FC</td>
<td>CE</td>
<td>EUO</td>
<td>CV mutex unowned</td>
</tr>
<tr>
<td>01</td>
<td>000006FE</td>
<td>CE</td>
<td>EIN</td>
<td>CV initialization failed (duplicate)</td>
</tr>
<tr>
<td>01</td>
<td>00000702</td>
<td>RW</td>
<td>R</td>
<td>Release</td>
</tr>
<tr>
<td>01</td>
<td>00000704</td>
<td>RW</td>
<td>AW</td>
<td>Acquire after wait</td>
</tr>
<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>
</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>00000717</td>
<td>RW</td>
<td>RA</td>
<td>Read acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000718</td>
<td>RW</td>
<td>WA</td>
<td>Write acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000719</td>
<td>RW</td>
<td>RB</td>
<td>Read busy (tryread failed)</td>
</tr>
<tr>
<td>01</td>
<td>0000071A</td>
<td>RW</td>
<td>WB</td>
<td>Write busy (trywrite failed)</td>
</tr>
<tr>
<td>01</td>
<td>0000071B</td>
<td>RW</td>
<td>RW</td>
<td>Read wait</td>
</tr>
<tr>
<td>01</td>
<td>0000071C</td>
<td>RW</td>
<td>WW</td>
<td>Write wait</td>
</tr>
<tr>
<td>01</td>
<td>0000071D</td>
<td>RW</td>
<td>BI</td>
<td>Call to SLK_INIT</td>
</tr>
<tr>
<td>01</td>
<td>0000071E</td>
<td>RW</td>
<td>BD</td>
<td>Call to SLK_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>0000071F</td>
<td>RW</td>
<td>BO</td>
<td>Call to SLK_OBTAIN</td>
</tr>
<tr>
<td>01</td>
<td>00000720</td>
<td>RW</td>
<td>BC</td>
<td>Call to SLK_OBTAIN_COND</td>
</tr>
<tr>
<td>01</td>
<td>00000721</td>
<td>RW</td>
<td>BR</td>
<td>Call to SLK_RELEASE</td>
</tr>
<tr>
<td>01</td>
<td>000008DC</td>
<td>RE</td>
<td>EOW</td>
<td>Error - Already owned for write (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008DD</td>
<td>RE</td>
<td>EOR</td>
<td>Error - Already owned for read (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008DE</td>
<td>RE</td>
<td>EDL</td>
<td>Error - BPX1SLK(fail) returns EDEADLK</td>
</tr>
<tr>
<td>01</td>
<td>000008DF</td>
<td>RE</td>
<td>EIV</td>
<td>Error - BPX1SLK(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000008E0</td>
<td>RE</td>
<td>EPM</td>
<td>Error - BPX1SLK(fail) returns EPERM</td>
</tr>
<tr>
<td>01</td>
<td>000008E1</td>
<td>RE</td>
<td>EAG</td>
<td>Error - BPX1SLK(fail) returns EAGAIN</td>
</tr>
<tr>
<td>01</td>
<td>000008E2</td>
<td>RE</td>
<td>EBS</td>
<td>Error - BPX1SLK(fail) returns EBUSY</td>
</tr>
<tr>
<td>01</td>
<td>000008E3</td>
<td>RE</td>
<td>ENM</td>
<td>Error - BPX1SLK(fail) returns ENOMEM</td>
</tr>
<tr>
<td>01</td>
<td>000008E4</td>
<td>RE</td>
<td>EBR</td>
<td>Error - BPX1SLK(RELEASE) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E5</td>
<td>RE</td>
<td>EBC</td>
<td>Error - BPX1SLK(OBTAIN_COND) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E6</td>
<td>RE</td>
<td>EBO</td>
<td>Error - BPX1SLK(OBTAIN) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E7</td>
<td>RE</td>
<td>EBD</td>
<td>Error - BPX1SLK(DESTROY) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E8</td>
<td>RE</td>
<td>EBI</td>
<td>Error - BPX1SLK(INIT) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E9</td>
<td>RE</td>
<td>EFR</td>
<td>Error - Forced release</td>
</tr>
<tr>
<td>01</td>
<td>000008EA</td>
<td>RE</td>
<td>EFD</td>
<td>Error - Forced decrement</td>
</tr>
<tr>
<td>01</td>
<td>000008ED</td>
<td>RE</td>
<td>EIA</td>
<td>Error - Initialization failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000008EE</td>
<td>RE</td>
<td>EIS</td>
<td>Error - Initialization failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000008EF</td>
<td>RE</td>
<td>EF</td>
<td>Error - Forced by quiesce</td>
</tr>
<tr>
<td>01</td>
<td>000008F0</td>
<td>RE</td>
<td>EP</td>
<td>Error - Program check</td>
</tr>
<tr>
<td>01</td>
<td>000008FB</td>
<td>RE</td>
<td>EU1</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>

The format for the Mutex – Condition Variable – Latch entries in the trace table is:

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>

Chapter 12. Using Language Environment AMODE 64 debugging facilities 471
Where each field represents:

**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 31 on page 467.

**Object address**  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. Currently this is only supported by C/C++.

**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>00000005</td>
<td>Storage allocation entry</td>
</tr>
<tr>
<td>03</td>
<td>00000006</td>
<td>Storage allocation exit</td>
</tr>
</tbody>
</table>

For a detailed description of these records, see "C/C++ contents of the Language Environment trace tables" on page 498.

**Sample dump for the trace table entry**

The following is an example of a dump of the trace table when you specify the LE=1 suboption (the library call/return trace):
Language Environment Trace Table:
Most recent trace entry is at displacement: 00000000

<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 22.02.389659 Date 2004.04.08 Thread ID... 2548146000000001</td>
<td>main --&gt;(006F) printf()</td>
</tr>
<tr>
<td>+000018</td>
<td>9481B995 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000038</td>
<td>6606E4AD 0060F6C6 50409799 8954388 40404040 40404040 40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000058</td>
<td>40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040</td>
<td></td>
</tr>
<tr>
<td>+000078</td>
<td>40404040 40404040</td>
<td></td>
</tr>
</tbody>
</table>

| Time 22.02.389724 Date 2004.04.08 Thread ID... 2548146000000001 | <--(006F) R1=000000000000000 R2=000000000000000
| +000080 | 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 | |
| +001118 | 9481B995 40404040 40404040 40404040 40404040 | |
| +001338 | 6606E4AD 0060F6C6 50409799 8954388 | |
| +001558 | 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 | |
| +001778 | 40404040 40404040 | |

| Time 22.02.389904 Date 2004.04.08 Thread ID... 2548146000000001 | <--(0170) d11load() |
| +001800 | 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 | |
| +002118 | 9481B995 40404040 40404040 40404040 | |
| +002338 | 6606E4AD 0060F6C6 50409799 8954388 | |
| +002558 | 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 | |
| +002778 | 40404040 40404040 | |

| Time 22.02.389938 Date 2004.04.08 Thread ID... 2548146000000001 | <--(006F) printf() |
| +002800 | 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 | |
| +002918 | 9481B995 40404040 40404040 40404040 40404040 40404040 | |
| +003138 | 6606E4AD 0060F6C6 50409799 8954388 | |
| +003358 | 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 | |
| +003578 | 40404040 40404040 | |

---

**Figure 153. Trace table in dump output**

### 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 the signal SIGDUMP to the same processes in order to capture the trace output. See the **UNIX System Services Command Reference** for more information about the SIGTRACE signal.

---

Chapter 12. Using Language Environment AMODE 64 debugging facilities 473
Chapter 13. Debugging AMODE 64 C/C++ routines

This chapter 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 which 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)

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 154 on page 477 shows the structure as it appears in stdio.h.
typedef struct __amrctype {
  union {
    int __error;
    struct {
      unsigned short __syscode,
      __rc;
    } __abend;
    struct {
      unsigned char __fdbk_fill,
      __rc,
      __ftncd,
      __fdbk;
    } __feedback;
    struct {
      unsigned short __svc99_info,
      __svc99_error;
    } __alloc;
    unsigned char __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
  #if __EDC_TARGET >= 0x41080000
  #ifdef __LP64
  unsigned long __XRBA; /* 8 byte RBA */
  #elif defined(__LL)
  unsigned 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
  /* QSAM to BSAM switch reason */
  char __amrc_noseek_to_seek;
  /* padding to make amrc 256 bytes */
  char __amrc_pad[23];
} __amrc_type;

Figure 154. __amrc structure

Figure 155 shows the __amrc2 structure as it appears in stdio.h.

struct {
  int __error2;
  char __pad_error2[4];
  FILE *__fileptr;
  int __reserved(6);
}

Figure 155. __amrc2 structure
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.

__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.

__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.

__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.

__alloc

A structure that contains errors during fopen or freopen calls when defining files to the system using SVC 99.

__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).

__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 34 on page 479.

__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.

__rplfdbwd

This field contains feedback information related to a VSAM RLS failure. This is the feedback code from the IFGRPL control block.

__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().

__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>
</tbody>
</table>
Macro | Definition
--- | ---
__AM_BSAM_BSAMWRITE | The data set is already open for write (or update) in the same C process.
__AM_BSAM_FBS_APPEND | The data set is recfm=FBS and open for append
__AM_BSAM_LRECLX | The data set is recfm=LRECLX (used for VBS data sets where records span the largest blocksize allowed on the device)
__AM_BSAM_PARTITIONED_DIRECTORY | The data set is the directory for a regular or extended partitioned data set
__AM_BSAM_PARTITIONED_INDIRECT | The data set is a member of a partitioned data set, and the member name was not specified at allocation

[12] __error2 | A secondary error code. For example, an unsuccessful rename or remove operation places its reason code here.
[13] __fileptr | A pointer to the file that caused a SIGIOERR to be raised. Use an fdata() 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 34] lists __last_op values you could receive and where to look for further information.

**Table 34. __last_op values and diagnosis information**

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__IO_INIT</td>
<td>Will never be seen by SIGIOERR exit value given at initialization.</td>
</tr>
<tr>
<td>__BSAM_OPEN</td>
<td>Sets __error with return code from OS OPEN macro.</td>
</tr>
<tr>
<td>__BSAM_CLOSE</td>
<td>Sets __error with return code from OS CLOSE macro.</td>
</tr>
<tr>
<td>__BSAM_READ</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 66) filled in).</td>
</tr>
<tr>
<td>__BSAM_NOTE</td>
<td>NOTE returned 0 unexpectedly, no return code.</td>
</tr>
<tr>
<td>__BSAM_POINT</td>
<td>This will not appear as an error lastop.</td>
</tr>
<tr>
<td>__BSAM_WRITE</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 65) filled in).</td>
</tr>
<tr>
<td>__BSAM_CLOSE_T</td>
<td>Sets __error with return code from OS CLOSE TYPE=T.</td>
</tr>
<tr>
<td>__BSAM_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>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------</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>__IO_UNCATALOG</td>
<td>Sets __error with return code from I/O CAMLIST UNCAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>__IO_RENAME</td>
<td>Sets __error with return code from I/O CAMLIST 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 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_SI_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_UNEVEN</td>
<td>This occurs when an SI is written before the last double byte character is completed, thereby forcing C or C++ to fill in the last byte of the DBCS string with a padding byte X'FE'. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_CANNOT_EXTEND</td>
<td>This occurs when an attempt is made to extend a file that allows writing, but cannot be extended. Typically this is a member of a partitioned data set being opened for update.</td>
</tr>
<tr>
<td>__VSAM_OPEN_FAIL</td>
<td>Set when a low level VSAM OPEN fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_RRDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_KSDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS_PATH</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
</tbody>
</table>
### Table 34. `__last_op` values and diagnosis information (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__VSAM_OPEN_KSDS_PATH</code></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><code>__VSAM_MODCB</code></td>
<td>Set when a low level VSAM MODCB macro fails, sets <code>__rc</code> and <code>__fdbk</code> fields in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__VSAM_TESTCB</code></td>
<td>Set when a low level VSAM TESTCB macro fails, sets <code>__rc</code> and <code>__fdbk</code> fields in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__VSAM_SHOWCB</code></td>
<td>Set when a low level VSAM SHOWCB macro fails, sets <code>__rc</code> and <code>__fdbk</code> fields in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__VSAM_GENCB</code></td>
<td>Set when a low level VSAM GENCB macro fails, sets <code>__rc</code> and <code>__fdbk</code> fields in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__VSAM_GET</code></td>
<td>Set when the last op was a low level VSAM GET; if the GET fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__VSAM_PUT</code></td>
<td>Set when the last op was a low level VSAM PUT; if the PUT fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__VSAM_POINT</code></td>
<td>Set when the last op was a low level VSAM POINT; if the POINT fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__VSAM_ERASE</code></td>
<td>Set when the last op was a low level VSAM ERASE; if the ERASE fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__VSAM_ENDREQ</code></td>
<td>Set when the last op was a low level VSAM ENDREQ; if the ENDREQ fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__VSAM_CLOSE</code></td>
<td>Set when the last op was a low level VSAM CLOSE; if the CLOSE fails, sets <code>__rc</code> and <code>__fdbk</code> in the <code>__amrc</code> struct.</td>
</tr>
<tr>
<td><code>__QSAM_GET</code></td>
<td><code>__error</code> is not set (if abend (errno == 92), <code>__abend</code> is set, otherwise if read error (errno == 66), look at <code>__msg</code>.)</td>
</tr>
<tr>
<td><code>__QSAM_PUT</code></td>
<td><code>__error</code> is not set (if abend (errno == 92), <code>__abend</code> is set, otherwise if write error (errno == 65), look at <code>__msg</code>.)</td>
</tr>
<tr>
<td><code>__QSAM_TRUNC</code></td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td><code>__QSAM_FREEPOOL</code></td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td><code>__QSAM_CLOSE</code></td>
<td>Sets <code>__error</code> to result of OS CLOSE macro.</td>
</tr>
<tr>
<td><code>__QSAM_OPEN</code></td>
<td>Sets <code>__error</code> to result of OS OPEN macro.</td>
</tr>
<tr>
<td><code>__CMS_OPEN</code></td>
<td>Sets <code>__error</code> to result of FSOPEN.</td>
</tr>
<tr>
<td><code>__CMS_CLOSE</code></td>
<td>Sets <code>__error</code> to result of FSCLOSE.</td>
</tr>
<tr>
<td><code>__CMS_READ</code></td>
<td>Sets <code>__error</code> to result of FSREAD.</td>
</tr>
<tr>
<td><code>__CMS_WRITE</code></td>
<td>Sets <code>__error</code> to result of FSWRITE.</td>
</tr>
<tr>
<td><code>__CMS_STATE</code></td>
<td>Sets <code>__error</code> to result of FSSTATE.</td>
</tr>
<tr>
<td><code>__CMS_ERASE</code></td>
<td>Sets <code>__error</code> to result of FSERASE.</td>
</tr>
<tr>
<td><code>__CMS_RENAME</code></td>
<td>Sets <code>__error</code> to result of CMS RENAME command.</td>
</tr>
<tr>
<td><code>__CMS_EXTRACT</code></td>
<td>Sets <code>__error</code> to result of DMS EXTRACT call.</td>
</tr>
<tr>
<td><code>__CMS_LINERD</code></td>
<td>Sets <code>__error</code> to result of LINERD macro.</td>
</tr>
<tr>
<td><code>__CMS_LINEWRT</code></td>
<td>Sets <code>__error</code> to result of LINEWRT macro.</td>
</tr>
<tr>
<td><code>__CMS_QUERY</code></td>
<td><code>__error</code> is not set.</td>
</tr>
</tbody>
</table>
Table 34. __last_op values and diagnosis information (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__HSP_CREATE</td>
<td>Indicates last op was a DSPSERV CREATE to create a hiper-space for a hiper-space 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 hiper-space for a hiper-space 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 hiper-space. 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 hiper-space. 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 hiper-space. 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 <a href="https://www.ibm.com/support/knowledgecenter/en/sslfac_pdf/rzocics/">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="https://www.ibm.com/support/knowledgecenter/en/sslfac_pdf/rzocics/">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="https://www.ibm.com/support/knowledgecenter/en/sslfac_pdf/rzocics/">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</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 <a href="https://www.ibm.com/support/knowledgecenter/en/sslfac_pdf/rzocics/">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</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 <a href="https://www.ibm.com/support/knowledgecenter/en/sslfac_pdf/rzocics/">z/OS UNIX System Services Programming: Assembler Callable Services Reference</a>.</td>
</tr>
<tr>
<td>__LFS_FSTAT</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in <a href="https://www.ibm.com/support/knowledgecenter/en/sslfac_pdf/rzocics/">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

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 156 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 156. 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.

Figure 157 on page 484 is an example of a routine using __errno2() and Figure 158 on page 484 shows the sample output from that routine.
#pragma runopts(posix(on))
#define _EXT
#include <stdio.h>
#include <errno.h>

int main(void) {
    FILE *f;
    f = fopen("testfile.dat", "r")
    if (f==NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    }
    return 0;
}

Figure 157. Example of a routine using __errno2()
#pragma runopts(posix(on))
#define _EXT
#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\n", __errno2());
    }
    /* reset errno2 to zero */
    *__err2ad() = 0x0;
    printf("__errno2 = %08x\n", __errno2());
    f = fopen("testfile.dat", "r");
    if (fp == NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    }
    return 0;
}

Figure 161. Example of a routine using __err2ad() in combination with __errno2()

fopen() failed: EDC5129I No such file or directory.
__errno2 = 05620062
__errno2 = 00000000
fopen() failed: EDC5129I No such file or directory.
__errno2 = 05620062

Figure 162. Sample output of routine using __err2ad() in combination with __errno2()

For more information about _EDC_ADD_ERRNO2, see z/OS XL C/C++ Programming Guide.

For more information about __errno2() and __err2ad(), see z/OS XL C/C++ 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 `cdizero` shown in Figure 59 on page 214.

2. From the compiler listing, locate the variable in the storage offset listing:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Offset</th>
<th>Type</th>
<th>Location</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>5823-0:10</td>
<td>automatic</td>
<td>2248(r4)</td>
<td>4</td>
</tr>
</tbody>
</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 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>+00000016</td>
<td>DIVERZERO</td>
<td>Call</td>
<td></td>
<td></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 cdivzero shown in Figure 53. C routine with a divide-by-zero error.

2. From the compiler listing, locate the parameter in the storage offset listing:

   pp 5828-0:15  Class = parameter,  Location = 2432(r4),  Length = 8

3. From the Traceback (in the Language Environment dump or in the formatted output from the IPCS VERBEXIT LEDATA 'CEEDUMP' subcommand for a system dump) locate the function:

   DSA Entry E Offset Load Mod Program Unit Service Status
   00000003 funcb +00000002 CDIVZERO Exception

   DSA Addr E Addr PU Addr PU Offset Comp Date Attributes
   00000003 000000010B2FF080 000000025100108 0000000000000000 ******** 20040408 XPLINK EBCDIC IEEE

   If the base register is r4, the register 4 value is always the DSA address for the function.

   If the base register is not r4, the register value must be located from saved registers.

   If the Status field indicates Exception, use the saved registers from when the condition occurred. In the Language Environment dump, the saved registers can be found in the Condition information associated with the DSA address in the "Condition Information for Active Routines" section. In the formatted output from the IPCS VERBEXIT LEDATA 'CM' subcommand for a system dump, the saved registers can be found in the CIBH that has the DSA address as the value for the SV1 field.

   If the Status field indicates Call, use the saved registers from the DSA address that appears on the line above the function in the Traceback. In the Language Environment dump, the DSAs can be found in the "Control Blocks for Active Routines" section. In the formatted output from the IPCS VERBEXIT LEDATA 'STACK' subcommand for a system dump, the DSAs can be found in the "DSA backchain" section.

   Note: Some functions do not save all registers.

4. Add the register value to the offset of the parameter to obtain the address of the parameter. In the Language Environment dump, the contents of the parameter can be read in the DSA Frame section corresponding to the function that passed the parameter. For a system dump, use the IPCS LIST subcommand to display the storage where the parameter is located.

   The address for parameter pp is X'1082FF080' + X'980' = X'1082FFA00'.

**Steps for finding members of aggregates**

You can define aggregates in any of the storage classes or pass them as parameters to a called function. The first step is to find the start of the aggregate. You can compute the start of the aggregate as described in previous sections, depending on the type of aggregate used.

The aggregate map provided for each declaration in a routine can further assist in finding the offset of a specific variable within an aggregate. Structure maps are generated using the AGGREGATE compiler option. Figure 163 on page 488 shows an example of an aggregate.
Figure 164 shows an example of aggregate map.

To find the value of variable `tempTargetRef.addr`:
1. Locate the automatic variable `tempTargetRef` in the storage offset listing:
   ```c
   typedef struct {
       int asid;
       void *addr;
       asfAmodeType amode;
   } asfTargetRef;
   asfTargetRef tempTargetRef;
   ```
   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 164. The offset is 8. Add the offset from the Aggregate Map to the address of the `tempTargetRef` variable.
   The result is X'1082FDC0' (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 165 on page 490 shows a sample C routine that uses the cdump function to generate a dump.

Figure 170 on page 493 shows the dump output.
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void hsigfpe(int);
void hsigterm(int);
void atf1(void);

typedef int (*FuncPtr_T)(void);

int st1 = 99;
int st2 = 255;
int xcount = 0;

int main(void) {
  /*
   * 1) Open multiple files
   * 2) Register 2 signals
   * 3) Register 1 atexit function
   * 4) Fetch and execute a module
   */

  FuncPtr_T fetchPtr;
  FILE* fp1;
  FILE* fp2;
  int rc;
  fp1 = fopen("myfile.data", "w");
  if (!fp1) {
    perror("Could not open myfile.data for write");
    exit(101);
  }

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

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

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

  signal(SIGFPE, hsigfpe);
  signal(SIGTERM, hsigterm);

  rc = atexit(atf1);
  if (rc) {
    fprintf(stderr, "Failed on registration of atexit function atf1\n");
    exit(103);
  }
}

Figure 165. Example C routine using cdump() to generate a dump (Part 1 of 2)
fetchPtr = (FuncPtr_T) fetch("MODULE1");
if (!fetchPtr) {
    fprintf(stderr, "Failed to fetch MODULE1\n");
    exit(104);
} fetchPtr();
return(0);
}

void hsigfpe(int sig) {
  ++st1;
  return;
}

void hsigterm(int sig) {
  ++st2;
  return;
}

void atf1() {
  ++xcount;
}

Figure 165. Example C routine using cdump() to generate a dump (Part 2 of 2)

Figure 166 shows a fetched C module:

#include <ctest.h>
#pragma linkage(func1, fetchable)
int func1(void) {
  __cdump("This is a sample dump");
  return(0);
}

Figure 166. Fetched module for C routine

Sample C++ routine that generates a Language Environment dump

Figure 167 shows a sample C++ routine that uses a protection exception to generate a dump.

#include <iostream.h>
#include <ctest.h>
#include "stack.h"

int main() {
  cout << "Program starting:\n";
  cerr << "Error report:\n";
  Stack<int> x;
  x.push(1);
  cout << "Top value on stack : " << x.pop() << '\n';
  cout << "Next value on stack: " << x.pop() << '\n';
  return(0);
}

Figure 167. Example C++ routine with protection exception generating a dump
Figure 168 shows the template file stack.c

```c
#ifndef __STACK__
  #include "stack.h"
#endif

template <class T> T Stack<T>::pop() {
  T value = head->value;
  head = head->next;
  return(value);
}
template <class T> void Stack<T>::push(T value) {
  Node* newNode = new Node;
  newNode->value = value;
  newNode->next = head;
  head = newNode;
}
```

Figure 168. Template file STACK.C

Figure 169 shows the header file stack.h.

```c
#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 169. Header file STACK.H

Sample Language Environment dump with C/C++-specific information

This sample dump was produced by compiling the routines shown in Figure 165 on page 490 and Figure 166 on page 491. 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 166 on page 491. func1 calls the library routine _cdump() in statement number 6. The complete program unit names for main and func1 are shown in the Fully Qualified Names section along with its load module name.
Figure 170. Example dump from sample C routine (Part 1 of 4)
Figure 170. Example dump from sample C routine (Part 2 of 4)
Figure 170. Example dump from sample C routine (Part 3 of 4)
Chapter 13. Debugging AMODE 64 C/C++ routines

Figure 170. Example dump from sample C routine (Part 4 of 4)
C/C++ contents of the Language Environment trace tables

Language Environment provides four C/C++ trace table entry types that contain character data:

- Trace entry 5 occurs when a C library function is called.
- Trace entry 6 occurs when a C library function returns.

The format for trace table entry 5 is:

```
NameOfCallingFunction
-->(xxxx) 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.

The format for trace table entry 6 is:

```
<--(xxxx)
R1=xxxxxxxxxxxxxxxxx R2=xxxxxxxxxxxxxxxxx R3=xxxxxxxxxxxxxxxxx
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 171 on page 499 shows an XPLINK trace which has examples of the trace entries 5 and 6.
Figure 171. Trace table with XPLINK trace table entries 5 and 6. (Part 1 of 2)
Divide-by-zero error

Figure 172 on page 501 illustrates a C program that contains a divide-by-zero error. The code was compiled with RENT so static and external variables need to be calculated from the WSA field. The code was compiled with LP64, GONUM (to produce statement numbers) and XREF, LIST and OFFSET to generate a listing, which is used to calculate addresses of functions and data. The code was processed by the binder with MAP to generate a binder map, which is used to calculate the addresses of static and external variables. The program was created with the option TERMTHDACP(UADUMP) which produced both a Language environment dump and a system dump.

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.

Figure 171. Trace table with XPLINK trace table entries 5 and 6. (Part 2 of 2)

For more information about the Language Environment trace table format, see "Understanding the trace table entry (TTE)" on page 466.
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 173 on page 502 shows the generated traceback from the dump.

---

/* 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;
}
Information for enclave main

Information for thread 25AC70A0000000000

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>00000001</td>
<td>CEEDSP</td>
<td>+00000000</td>
<td>CELQLIB</td>
<td>CEEDSP</td>
<td>HLE773B</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>00000002</td>
<td>CEEOS1GJ</td>
<td>+0000094E</td>
<td>CELQLIB</td>
<td>CEEOS1GJ</td>
<td>HLE773B</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>00000003</td>
<td>CELQHODD</td>
<td>+0000024E</td>
<td>CELQLIB</td>
<td>CELQHODD</td>
<td>HLE773B</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>00000004</td>
<td>CEEOS1GG</td>
<td>-0D6A6E0</td>
<td>CELQLIB</td>
<td>CEEOS1GG</td>
<td>HLE773B</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>00000005</td>
<td>CELQHROD</td>
<td>+0000024E</td>
<td>CELQLIB</td>
<td>CELQHROD</td>
<td>HLE773B</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>00000006</td>
<td>funcb</td>
<td>+00000652</td>
<td>1B</td>
<td>CDIVZERO</td>
<td>CDIVZERO</td>
<td>Exception</td>
<td></td>
</tr>
<tr>
<td>00000007</td>
<td>main</td>
<td>+00000626</td>
<td>12</td>
<td>CDIVZERO</td>
<td>CDIVZERO</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>00000008</td>
<td>CELQINIT</td>
<td>+00001340</td>
<td>CELQLIB</td>
<td>CELQINIT</td>
<td>HLE773B</td>
<td>Call</td>
<td></td>
</tr>
</tbody>
</table>

DSA | DSA Addr | E Addr | PU Addr | PU Offset | Comp Date Attributes |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>000000001082FAC0</td>
<td>0000000000000000</td>
<td>0000000025793EB8</td>
<td>0000000025793EB8</td>
<td>00000000</td>
</tr>
<tr>
<td>00000002</td>
<td>000000001082FD3E0</td>
<td>0000000000000000</td>
<td>000000002588FE28</td>
<td>000000002588FE28</td>
<td>00000094E</td>
</tr>
<tr>
<td>00000003</td>
<td>000000001082FDE0</td>
<td>0000000000000000</td>
<td>00000000257850D8</td>
<td>00000000257850D8</td>
<td>000000024E</td>
</tr>
<tr>
<td>00000004</td>
<td>000000001082FE020</td>
<td>0000000000000000</td>
<td>00000000258894E0</td>
<td>00000000258894E0</td>
<td>0D6A6E0</td>
</tr>
<tr>
<td>00000005</td>
<td>000000001082FE40</td>
<td>0000000000000000</td>
<td>00000000257850D8</td>
<td>00000000257850D8</td>
<td>000000024E</td>
</tr>
<tr>
<td>00000006</td>
<td>000000001082FF080</td>
<td>0000000000000000</td>
<td>0000000025780B0D</td>
<td>0000000025780B0D</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>00000007</td>
<td>000000001082FF1B0</td>
<td>0000000000000000</td>
<td>0000000025708170</td>
<td>0000000025708170</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>00000008</td>
<td>000000001082FF2B0</td>
<td>0000000000000000</td>
<td>0000000025708010</td>
<td>0000000025708010</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Fully Qualified Names

<table>
<thead>
<tr>
<th>DSA</th>
<th>Entry</th>
<th>Program Unit</th>
<th>Load Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000006</td>
<td>funcb</td>
<td>PLPSC://POSIX.CRTL.C(CDIVZERO)'</td>
<td>/u/alifar/tools/CDIVZERO</td>
</tr>
<tr>
<td>00000007</td>
<td>main</td>
<td>PLPSC://POSIX.CRTL.C(CDIVZERO)'</td>
<td>/u/alifar/tools/CDIVZERO</td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Condition Information for PLPSC://POSIX.CRTL.C(CDIVZERO)' (DSA address 00000001082F9F00)

CIB Address: 00000001082F9F00

Current Condition:
CEE198S 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: 1B Offset: +00000052

Machine State:
ILC..... 0002 Interruption Code..... 0009
PSM..... 0785240100000000 0000000025708124
GP90.... 0000000000000000 GP91.... 000000001082F4A0 GP92.... 00000000108414340 GP93.... 00000000108414340
GP94.... 000000001082FF90 GP95.... 0000000010830050 GP96.... 0000000000000000 GP97.... 0000000000000000
GP98.... 0000000025708188 GP99.... 0000000025708278 GP990.... 0000000025708278 GP991.... 00000000108414340
GP912.... 000000000000005300 GP913.... 000000000000006E60 GP914.... 0000000025782F1B GP915.... 000000000000001F
FP1.... 0000000000000000
FP12.... 0000000000000000
FP13.... 0000000000000000
FP14.... 0000000000000000
FP15.... 0000000000000000

Storage dump near condition, beginning at location(0000000025708112)
+0000 0000000025708112 0004E300 70000014 A70BFF78 E600020
+0310 0000000025708122 1D60B904 00075000 48COE320 48C00014

Figure 173. Sections of the dump from example C/C++ routine (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 funcb is called. Similarly, statement number 18, corresponding to DSA 6, points to line:
\[ \text{result} = \frac{\text{fa}}{\text{statint}-73}; \] 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 174 on page 504.

   The offset (within funcb) of the exception from the traceback (X'52') reveals the divide instruction: DR R6,R0 at that location. Instructions at offsets X'32' through X'58' refer to the \( \text{result} = \frac{\text{fa}}{(\text{statint}-73)}; \) line of the C/C++ routine.
Figure 174. Pseudo assembly listing (Part 1 of 3)
*** General purpose registers used: 1111011101000000000
*** Floating point registers used: 1111111000000000000
*** Size of register spill area: 12B(max) 0(used)
*** Size of dynamic storage: 0
*** Size of executable code: 144

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 fncb(int *np);
000009 | int main(void) {

0000C8 00000000 | 01L0 DS 0D
0000C8 00C300C5 | +F'12779717' XPLink entrypoint marker
0000C8 000C500F1 | +F'12910833'
0000DD 00000090 | +F'144'
0000D4 00000100 | +F'256'
000000 | main DS 0D
000000 | 000009 STMG r5,r7,1800(r4)
000006 A748 FF00 | AGHI r4,H'-258'
00000A | End of Prolog

000010 | int aa, bb=1;
000012 5000 48C4 | 000010 LG HI r0,H'+1'
000012 5000 48C4 | 000011 ST r0,bb,(r4,2244)
000012 5000 48C4 | 000011 aa = bb;
000012 5000 48C4 | 000011 LG F r0,bb,(r4,2244)
000012 5000 48C4 | 000011 ST r0,aa,(r4,2240)
000012 5000 48C4 | aa = fncb(3);
000011 5000 48C4 | LA r1,aa,(r4,2240)
000020 E350 48B8 | 000011 LG F r5,Save_Addr_Ptr1,(r4,2056)
000026 A775 FF90 | 000012 BRAS r7,fncb
00002A 0701 | 000012 NOPR 1
00002C 0000003 | 000012 LGR r0,r3
000030 5000 48C0 | 000012 ST r0,aa,(r4,2240)
00003A E330 48C0 | 000013 return(aa);
00003A E330 48C0 | 000013 LG F r3,aa,(r4,2240)
00003D | } int
00003A | 000014 01L2 DS 0H
00003A | Start of Epilog

00003A E370 4818 | 000014 LG r7,2072,(r4)
000040 4140 4100 | 000014 LG r4,256,(r4)
000044 47F0 7002 | 000014 B 2,(r7)

*** General purpose registers used: 1111011101000000000
*** Floating point registers used: 1111111100000000000
*** Size of register spill area: 12B(max) 0(used)
*** Size of dynamic storage: 0
*** Size of executable code: 72

Constant Area

000000 D985A2AA 93A3407E 406C8415 00 | Result = %d..

PPA1: Entry Point Constants

000000 02 | =AL1(2) Version
000001 CE | =AL1(206) CEL signature
000002 07C0 | =H'1984' GPR save mask
000004 00000090 | =A(PPA2-PPA1)
000008 00002081 | =F'2139694399'
00000C 0002 | =H'2'
00000E 0503 | =H'1283' Prol len/2; alloca reg; R4 change offset/2
000010 00000090 | =F'144'
000014 01000000 | =F'16777216'
000018 0005 **** | =AL2(5),C'fncb'
000020 FFFFFEF8 | =F'-264'

PPA1 End

Figure 174. Pseudo assembly listing (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 175 shows the WSA address.

```
<table>
<thead>
<tr>
<th>DLL Information:</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSA Addr</td>
</tr>
<tr>
<td>0000000108300050</td>
</tr>
</tbody>
</table>
```

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 176 on page 507. In this example, the offset is `X'30'`.

![Figure 174. Pseudo assembly listing (Part 3 of 3)](image-url)
6. Add the WSA address of X'108300050' to the offset of statint. The result is X'108300080'. This is the address of the variable statint, which is in the writable static area.

7. Use IPCS to display the writeable static area in the system dump. The value at location X'108300080' is X'49' (that is, statint is 73), and hence the fixed-point divide exception.

---

**Figure 176. Writable static map**

6. Add the WSA address of X'108300050' to the offset of statint. The result is X'108300080'. This is the address of the variable statint, which is in the writable static area.

7. Use IPCS to display the writeable static area in the system dump. The value at location X'108300080' is X'49' (that is, statint is 73), and hence the fixed-point divide exception.

---

**Figure 177. IPCS storage display of the writeable static area**

---

**Calling a nonexistent function**

[Figure 178 on page 508] 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 179 on page 509. In this example, the message is CEE3201S. The system detected an operation exception (System Completion Code=OC1). 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 178. 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.
Figure 179. Sections of the dump from example C routine (Part 1 of 3)
Figure 179. Sections of the dump from example C routine (Part 2 of 3)
2. Find the branch instructions for funca in the listing in Figure 180 on page 512. 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 178 on page 508.
Figure 180. Pseudo assembly listing (Part 1 of 2)
3. Find the offset of func_ptr in the Writable Static Map, shown in Figure 181 on page 514
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 180 on page 512). 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 181 shows the sections of the dump.

---

**Figure 181. Writable static map**

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>NAME</th>
<th>TYPE</th>
<th>LENGTH</th>
<th>SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$PRIV000012</td>
<td>PART</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>EXIST#S</td>
<td>PART</td>
<td>30</td>
<td>EXIST#C</td>
</tr>
<tr>
<td>40</td>
<td>func_ptr</td>
<td>PART</td>
<td>8</td>
<td>func_ptr</td>
</tr>
</tbody>
</table>

---

**Figure 181. Writable static map**

**Figure 182. IPCS storage display of the writeable static area**

### Handling dumps written to the z/OS UNIX file system

When a z/OS UNIX C/C++ application program is running in an address space created as a result of a call to `spawn()`, `vfork()`, or one of the exec family of functions, the SYSMDUMP DD allocation information is not inherited. Even though the SYSMDUMP allocation is not inherited, a SYSMDUMP allocation must exist in the parent in order to obtain a HFS storage dump. If the program terminates abnormally while running in this new address space, the kernel causes an unformatted storage dump to be written to an HFS file in the user's working directory. The file is placed in the current working directory or into /tmp if the current working directory is not defined. The file name has the following format:

```
/directory/coredump.pid
```

where directory is the current working directory or tmp, and pid is the hexadecimal process ID (PID) for the process that terminated. For details on how to generate the system dump, see "Steps for generating a system dump in a z/OS UNIX shell" on page 396.

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. 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

<table>
<thead>
<tr>
<th>General Data</th>
<th>Current Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management class</td>
<td>STANDARD</td>
</tr>
<tr>
<td>Allocated cylinders</td>
<td>30</td>
</tr>
<tr>
<td>Storage class</td>
<td>05390</td>
</tr>
<tr>
<td>Allocated extents</td>
<td>1</td>
</tr>
<tr>
<td>Volume serial</td>
<td>DPXDUI</td>
</tr>
<tr>
<td>Device type</td>
<td>3380</td>
</tr>
<tr>
<td>Data class</td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>PS</td>
</tr>
<tr>
<td>Current Utilization</td>
<td></td>
</tr>
<tr>
<td>Record format</td>
<td>FB</td>
</tr>
<tr>
<td>Used cylinders</td>
<td>0</td>
</tr>
<tr>
<td>Record length</td>
<td>4160</td>
</tr>
<tr>
<td>Used extents</td>
<td>0</td>
</tr>
<tr>
<td>Block size</td>
<td>4160</td>
</tr>
<tr>
<td>1st extent cylinders</td>
<td>30</td>
</tr>
<tr>
<td>Secondary cylinders</td>
<td>10</td>
</tr>
<tr>
<td>Data set name type</td>
<td></td>
</tr>
</tbody>
</table>

Creation date : 2001/08/30
Expiration date : ***None***

---

**Figure 183. 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 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 397 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:

- 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.
Details on how to request and interpret the reports are provided in the following sections.

**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 report showing HeapPools statistics for a multithreaded C/C++ application, see Figure 139 on page 362.

The following describes the C/C++ specific heap pool information.

**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.

**Usage 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: $gggg$
  - $p$ — 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 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 Per Extent — 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.

Note: 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).

Note: The suggested cell sizes are given with no cell counts because the usage of each new cell pool size is not known. If there are less than 12 cell sizes calculated, then the last pool size is set at 65536.

For more information about stack and heap storage for AMODE64 applications, see z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode.

HEAPPOOLS storage statistics
The HEAPPOOLS 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.

Usage 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:
Initial Heap Size * (Extent Percent/100)/(Element Size) with a minimum of four cells.

**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.
  In order 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.

**Note:** 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:
  \[
  \frac{(\text{Maximum Cells Used} \times \text{Element Size} \times 100)}{\text{Initial Heap Size}} \times 0.01
  \]
  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](https://www.ibm.com/support/docview.wss?uid=swg21230411).

**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.

- 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.

**Note:** 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).

**Note:** 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.


A sample storage report generated by `__uheapreport()` is shown in [Figure 184 on page 520](https://www.ibm.com/support/knowledgecenter/SSLTBK_22.0.0/com.ibm.zos.v2r11.sdk.extend.doc_2201/cphIntro.html).
Language Environment V01 R11.00

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>Cells Per Extent</th>
<th>Extents</th>
<th>Maximum Cells Allocated</th>
<th>Maximum Cells Used</th>
<th>Maximum Cells In Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>128</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>512</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>2048</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>8192</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16384</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

Suggested Cell Sizes:
- 32, 128, 512, 2048, 8192, 16384, 0

End of Storage Report

Figure 184. Storage report generated by __uheapreport()
Appendix A. Diagnosing problems with Language Environment

This appendix 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 375.
   - 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 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>
</tbody>
</table>
### Entry point type is... | If...
---|---
Nonconforming | 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.

For routines with Language Environment-conforming and C/C++ entry points, Language Environment provides program prolog areas (PPAs). PPA1 contains the entry point name and the address of the PPA2; PPA2 contains pointers to the timestamp, where release level keyword information is found, and to the PPA1 associated with the primary entry point of the routine.

If the entry point type of the failing routine is Language Environment-conforming, go to step 3.
If the entry point type is C/C++, go to step 5.
If the entry point type is nonconforming, go to step 6 on page 524.

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 illustrations and 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.
   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.

   [Figure 185 on page 524](#) illustrates the C PPA1.
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 186 illustrates 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.
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)


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 35 describes how to produce documentation required for submission with the APAR.

Table 35. 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

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to z/OS TSO/E Primer, z/OS TSO/E User’s Guide, and z/OS ISPF User’s Guide Vol I for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at: http://www.ibm.com/systems/z/os/zos/bkserv/
Notices

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

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

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

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

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

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

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

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

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

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

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs
and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

If you are viewing this information softcopy, the photographs and color illustrations may not appear.
Policy for unsupported hardware

Various z/OS elements, such as DFSMS™, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

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).

IEEE is a trademark of the Institute of Electrical and Electronics Engineers, Inc. in the United States and other countries.

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

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and other countries.

Other company, product, and service names may 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, V4R1
- [Enterprise COBOL for z/OS Licensed Program Specifications](#) GI11-7871
- [Enterprise COBOL for z/OS Customization](#) SC23-8526
- [Enterprise COBOL for z/OS Language Reference](#) SC23-8528
- [Enterprise COBOL for z/OS Programming Guide](#) SC23-8529
- [Enterprise COBOL for z/OS Migration Guide](#) GC23-8527

### Enterprise COBOL for z/OS, V3R4
- [Enterprise COBOL for z/OS Licensed Program Specifications](#) GC27-1411
- [Enterprise COBOL for z/OS Customization](#) GC27-1410
- [Enterprise COBOL for z/OS Language Reference](#) SC27-1408
- [Enterprise COBOL for z/OS Programming Guide](#) SC27-1412
- [Enterprise COBOL for z/OS Migration Guide](#) GC27-1409

### COBOL for OS/390 & VM
- [COBOL for OS/390 & VM Licensed Program Specifications](#) GC26-9044
- [COBOL for OS/390 & VM Customization under OS/390](#) GC26-9045
- [COBOL for OS/390 & VM Language Reference](#) SC26-9046
- [COBOL for OS/390 & VM Programming Guide](#) SC26-9049
- [COBOL for OS/390 & VM Compiler and Run-Time Migration Guide](#) GC26-4764
Debug Tool
- Debug Tool documentation is available at: http://www.ibm.com/software/awdtools/debugtool/

VS FORTRAN Version 2
- Language Environment for MVS & VM Fortran Run-Time Migration Guide, SC26-8499
- Language and Library Reference, SC26-4221
- Programming Guide for CMS and MVS, SC26-4222

Enterprise PL/I for z/OS
- Enterprise PL/I for z/OS Licensed Program Specifications, GC27-1456
- Enterprise PL/I for z/OS Programming Guide, SC27-1457
- Enterprise PL/I for z/OS Language Reference, SC27-1460
- Enterprise PL/I for z/OS Migration Guide, GC27-1458
- Enterprise PL/I for z/OS Messages and Codes, SC27-1461

PL/I for MVS & VM
- PL/I for MVS & VM Licensed Program Specifications, GC26-3116
- PL/I for MVS & VM Programming Guide, SC26-3113
- PL/I for MVS & VM Language Reference, SC26-3114
- PL/I for MVS & VM Reference Summary, SX26-3821
- PL/I for MVS & VM Compiler and Run-Time Migration Guide, SC26-3118
- PL/I for MVS & VM Installation and Customization under MVS, SC26-3119
- PL/I for MVS & VM Compile-Time Messages and Codes, SC26-3229
- PL/I for MVS & VM Diagnosis Guide, SC26-3149

High Level Assembler for MVS & VM & VSE

Related Publications

CICS
- CICS Transaction Server for z/OS Installation Guide, GC34-6224
- CICS Operations and Utilities Guide, SC34-6229
- CICS Problem Determination Guide, SC34-6239
- CICS Resource Definition Guide, SC34-6228
- CICS Application Programming Guide, SC34-6231
- CICS Application Programming Reference, SC34-6232
- CICS System Definition Guide, SC34-6226

DB2
- Database 2 Application Programming and SQL Guide, SC26-4377

DFSMS/MVS
- z/OS MVS Program Management: User’s Guide and Reference, SA22-7643
- z/OS DFSMS DFM Guide and Reference, SC26-7395

IPCS
- z/OS MVS IPCS User’s Guide, SA22-7596
- z/OS MVS IPCS Commands, SA22-7594
- z/OS MVS IPCS Customization, SA22-7595

DFSORT
- z/OS DFSORT Application Programming Guide, SC26-7523
IMS/ESA

IMS Version 8: Application Programming: 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 168, 476
__alloc 168, 476
__amrc 168, 476
__cabend() function 367, 374, 395
__code 168, 476
__error 168, 476
__feedback 168, 476
__last_op 168, 476
__le_cib_get() function 367
__le_message_get_and_and_write() function 360, 369
__le_message_get() function 369
__le_msg_write() function 360, 369
__msg 169, 478
__reset_exception_handler() function 367
__set_exception_handler() function 367, 368
_BPXK_MDUMP 83, 396
_EDC_ADD_ERRNO2 174, 484

Numerics
64-bit applications 359, 371, 375, 475

A
abend codes
>= 4000 30, 372
< 4000 30, 372
4093 371
  passing to operating system 23
  system, example of 32, 373
  user-specified, example of 32, 373
  user, example of 32, 373
  using 32, 373
abends
  internal, table of output 353
  Language Environment 32, 349, 373
  requested by assembler user exit 23
  system, example of 32, 374
  under CICS 349
  user 32, 374
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 529
AGGREGATE compiler option 4, 6
AMODE 64 applications
  classifying errors 371
  debugging C/C++ 475
  preparing for debugging 359
  using debugging 375
anywhere heap
  statistics 18
APAR (Authorized Program Analysis Report) 525

© Copyright IBM Corp. 1991, 2009

APAR (Authorized Program Analysis Report) (continued)
documentation 526
application program interfaces (API) 367, 369
application programs
debugging
  handling a storage dump written to a BFS file 224, 514
  handling a storage dump written to an HFS file 224, 514
argument
  in dump 62
arguments, registers, and variables for active routines 60, 393
assembler language
  user exit 23, 24
  for CICS 353
  generating a system dump with 81, 395
  modifying condition handling behavior and 23
  using 23, 24
atexit
  information in dump 197
Authorized Program Analysis Report (APAR) 525
automatic variables
  locating in dump 297, 330

B
base locator
  for working storage 246
  in dump 246
below heap
  statistics 18
BLOCKS option of CEE3DMP callable service 37

C
C library function
  trace table entries for 498
C return codes to CICS 350
C-CAA (C-specific common anchor area)
  See C/C++, C-specific common anchor area (C-CAA)
C/C++
  __amrc
    example of structure 168, 476
    information in dump 199
  __msg 169, 478
  atexit
    information in dump 197
C-specific common anchor area (C-CAA) 197
cdump() function 185, 488
compiler options 3
debugging examples 214, 221, 500
dump
  information in 198
  parameter in 181
  signal information in 197
C/C++ (continued)
structure variables, locating in 183
system, structures in 183
file
control block information 198
status and attributes in dump 198
functions
calling dump, example 185, 488
cdump() 43, 185, 377, 488
csnaph() 43, 185, 186, 377, 488, 489
ctrace() 43, 185, 186, 377, 488, 489
fetch() 167, 475
fopen() 169, 478
perror() 167, 174, 475, 483
printf() 168, 476
to produce dump output 43, 377
memory file control block 198
stdio.h 168, 476
timestamp 185
CAA (common anchor area) 65, 393, 398, 399, 401, 415, 454
call chain 62
CALL statement
CDUMP/CPDUMP 267
DUMP/PDUMP 266
SDUMP 268
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
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 26, 369
case 2 condition token 26, 369
cdump() function 185, 372, 488
CEE prefix 29, 31, 371, 373
CEE3ABD—terminate enclave with an abend 20, 32, 81
generating a dump and 81
handling user abends and 20, 32
modifying condition handling behavior and 20
CEE3DMP—generate dump 35, 62
See also Language Environment dump
generating a Language Environment dump with 35 options 36
relationship to PLIDUMP 289, 325
syntax 36
CEE3GRO—returns location offset 20
CEE3SRP—set resume point 20
CEE3XRITA assembler user exit 23
CEE3XRITA assembler user exit 353
CEEDCOD—decompose a condition token 26
CEEDUMP—Language Environment Dump Service
See Language Environment dump
control blocks 103, 413
locating 153
CEE3XRITA—register user condition handler 22
CEE3XRITA—register user exception handler 368
CEE3XRITA—get a message 26
CEE3XRITA—dispatch a message 24
CEEMRCE—move resume cursor to designated label 20
CEEMRCR—move resume cursor relative to handle cursor 20
CEE3XRITA—get a message, format, and dispatch a message 26
CEE3XRITA—signal a condition 26
CEE3XRITA—start 167
CE3XRITA prefix 371
CE3XRITAQSTRT 371
character
data dump 267
CHECK run-time option
function 8
modifying condition handling behavior and 21
CHECKOUT compiler option 4
CICS
abends 349
application, from an EXEC CICS command 354
debugging for 347
debugging information, table of locations 347
destination control table (DCT) 347
example traceback in CESE transient data queue 347
examples of output 347
nonzero reason code returned, table of output 353
reason codes 349
register and program status word contents 349
return codes
Language Environment 350
run-time messages 347
transaction
dump 348
rollback 353
class test 238
classifying errors table 29, 371
克莱 (COBOL load list entry) 245
COBCOM control block 250
COBOL
  base locator for working storage  246
  compiler options  5
  debugging examples  250, 263
dump
  external data in  246
  file information in  246
  linkage section in  246
  local variables in  243
  routine information in  243
  run unit storage in  248
  stack frames for active routines in  243
  working storage in  246
events  237
  listings  241
  memory file control block  197
  program class storage  246
  return codes to CICS  350
  routine
    calling Language Environment dump service  241
COBVEC control block  250
command
  syntax diagrams  xix
COMMAREA (Communication Area)  349
compiler options
  C  3
  COBOL  5
  Fortran  5
  LP64  359, 475
  PL/I  6
condition
  information
    for active routines  60, 392
    in dump  60, 392
  POSIX  26
  unhandled  27, 370
condition handling
  behavior, modifying  20
  user-written condition handler  20, 22
condition information block (CIB)  62, 74
condition manager  27, 370
CONDITION option of CEE3DMP callable service  38, 62
condition token  25, 26, 368
  case 1  26, 369
  case 2  26, 369
  example of  27, 369
  conditions, nested  27, 370
control block
  for active routines  61, 393
  csnap() function  186, 489
  ctrace() function  186, 372, 489
DCB (data control block)  245
DCT (destination control table)  347
DEBUG run-time option  8
downloading
  C, examples  214, 221, 500
  COBOL, examples  250, 263
  for CICS  347
  Fortran, examples  273, 279
  PL/I, examples  301, 307, 334, 342
  tool  35, 375
DEPTHCONDLMT run-time option
  function  8, 360
  modifying condition handling behavior and  21
  wait/loop error and  30
diagnosis checklist  521
disability  529
display
  errnojr_value  33
  DISPLAY statement  25, 237
displays
  errnojr_value  33
divide-by-zero error  214, 256, 275, 307, 312, 342, 346, 500
DSA (dynamic save area)  63
  See stack, frame
dummy DSA  63
dump
  an area of storage  267
  date in  56, 390
  dynamically allocated storage in  62
  storage
    written to an HFS file  224, 514
    symbolic  268
  DUMP suboption of TERMTHDACT run-time option  39, 375
  DUMP/PDUMP routine  266
    format specifications  266
    output  266
    usage considerations  266

E
ecb (enclave control block)  61, 393
edb (enclave data block)  61, 394
edc prefix  29, 31, 371, 373
edcontext shell command  33
eib (exec interface block)  349
enclave
  identifier in dump  58, 390
  member list  61, 62, 394
  storage  62
  termination
    behavior, establishing  24
  entry information  56, 389
ENTRY option of CEE3DMP callable service  38
entry point
  name of active routines in dump  391
errcount run-time option
  function  8
  modifying condition handling behavior and  21
  wait/loop error and  30
errno  197
ernojr_value
  displaying  33

data
  map listing  241
  values  62

Index  541
error
determining source of 521
message while Language Environment was handling
another error 27, 370
unanticipated 29, 371
ESD compiler option 6
examples
application abends from 354
C routines 214, 221, 500
calling a nonexistent subroutine 253, 305, 338
COBOL routines 250, 263
divide-by-zero error 214, 256, 275, 307, 312, 342, 346, 500
Fortran routines 273, 279
output under CICS 347
PL/I routines 301, 307, 334, 342
SUBSCRIPTRANGE error 250, 301, 334
exception handling
behavior, modifying 367
user-written exception handler 367
EXEC CICS DUMP statements 349
external data
for COBOL programs in dump 246

F
fetch
fetch information in dump 198
fetch() function 167, 475
fetchable module 167, 475
file
for COBOL, in dump 246
status key 238
file control block (FCB) 198
FILES option of CEE3DMP callable service 37
FLAG compiler option 4
floating point registers
in dump 57
fopen() function 169, 478
FOR prefix 29, 31
Fortran
compiler options 5
debugging examples 273, 279
dump services 265
ers, determining the source of 263
listings 265

G
general purpose registers 57
GMAREA 245
GONUMBER compiler option 4

H
HANDLE ABEND EXEC CICS command 347
header files, C
ctest.h 185, 488
errho.h 214, 500
stdio.h 168, 476
stdlib.h 214, 500

heap storage
created by CEECRHP callable service 19
in LEDATA Output 107, 418
reports 111, 427
storage in dump 62
user 17
HEAP64 run-time option 361
HEAPCHK run-time option
function 8, 112, 360, 428
HEAPPOOLS
storage statistics 226, 227, 517, 518
user-created, __uheapreport() 518
user-created, _uheapreport 227
trace report 87, 118, 399, 435
HEAPPOOLS run-time option 361
HEAPPOOLS64
storage statistics 516
HEAPPOOLS64 run-time option 361

I
I/O
conventions 167, 475
IBM prefix 29, 31
IGZ prefix 29, 31
INFOMSGFILTER run-time option
function 8, 360
INITIALIZE statement 238
instance specific information (ISI) 370
instruction length counter (ILC) in dump 60, 393
Inter-procedural Analysis (IPA) 359
interactive problem control system (IPCS)
analyzing a storage dump 225, 514
cbf command 462
Verbexit 397, 398, 401, 442
INTERRUPT compiler option
function 7
INTERRUPT run-time option 8
interruption code in dump 60, 393
IOHEAP64 run-time option 361
ITBLK in dump 250

K
keyboard 529

L
LAA (library anchor area) 463
language constructs 237
Language Environment
return codes to CICS 350
symbolic feedback code 25, 368
Language Environment dump 375, 377
C information in 196
CEEDUMP 8, 35, 359
COBOL information in 245
default options 38
example traceback in 62, 394
Fortran information in 271
multiple enclaves and 80
Language Environment dump (continued)

options
- BLOCKS 37
- CONDITION 38, 62
- ENCLAVE 36
- 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, 62
- VARIABLES 37, 62

output
- for C routines 190
- for COBOL program 241
- for Fortran routines 271
- for PL/I routines 289, 324

information for multiple enclaves 43, 377

PL/I information in 291, 327

section descriptions 56, 389

TERMTHDACT suboptions 40, 376
title 56, 389

traceback with condition information
- C routine 190, 492
- COBOL program 243
- Fortran routine 264, 271, 272
- Language Environment routine 56, 389
- PL/I routine 289, 324

using C functions 43, 377

using CDUMP/CMDUMP subroutine 265

using CEE3DMP callable service 35, 56, 389

using DUMP/PDUMP subroutine 265

using PLINE3M subroutine 43, 289, 324

using SDUMP subroutine 265

using TERMTHDACT run-time option 39, 375

LCA (library communication area) 463

LEDATA
- IPCS Verbexit 85, 397
- C/C++ Output 123, 442
- COBOL Output 147
- Format 86, 398
- Parameters 86, 398
- Understanding Output 89, 401

LIBHEAP64 run-time option 361

linkage section
- for COBOL programs in dump 246

LIST compiler option 4, 5, 7

listings generated by compiler
- COBOL 241
- Fortran 265
- PL/I 282, 318

LMESSAGE compiler option 7

local
- variables 60

LP64
- compiler option 359, 475

M

machine state information
- in dump 60, 393

mainframe
- education xxi

MAP compiler option 5, 7

MEMLIMIT storage parameter 371

memory file control block (MFCB) 197, 198

message
- classifying errors and 30, 372
- run-time, CICS 347
- user-created 24
- using in your routine 24

module
- fetchable 167, 475

module name prefixes, Language Environment 29, 371

MSG suboption
- of TERMTHDACT 39, 375

MSGFILE run-time option
- function 8
- run-time messages and 30, 372

MSGQ run-time option 8

N

nested condition 27, 370

no response (wait/loop) 30, 372

NOBLOCKS option of CEE3DMP callable service 37

NOCONDITION option of CEE3DMP callable service 38

NOENTRY option of CEE3DMP callable service 38

NOFILES option of CEE3DMP callable service 37

NOTORSTORAGE option of CEE3DMP callable service 37

NOTRACEBACK option of CEE3DMP callable service 37

NOVARIABLES option of CEE3DMP callable service 37

OFFSET compiler option 4, 5, 7

optimizing
- C 3, 63
- COBOL 5
- Fortran 268
- PL/I 6

options
- C compiler 3
- COBOL compiler 5
defaults for dump 40, 376
determining run-time in effect 9, 11, 360

Fortran compiler 5

Language Environment run-time 8, 359
options (continued)
PL/I compiler 6, 7
out-of-storage condition
virtual storage 371
OUTDD compiler option 5
output
incorrect 30, 372
missing 30, 372

P
page number
in dump 56, 390
PAGESIZE(n) option of CEE3DMP callable service 38
parameter
checking value of 23
perror() function 174, 483
PL/I
address of interrupt, finding in dump 294, 329
CAA address, finding in dump 300, 333
common anchor area (CAA) 300, 333
compiler listings
object code listing 286
static storage map 285
variable storage map 286
compiler options
generating listings with 282, 318
list of 6
CSECT 285
dumping examples 301, 307, 334, 342
dump
error type, finding in 294, 328
parameter list, finding contents in 298, 331
PL/I information, finding in 291, 298, 327, 331
PLIDUMP subroutine and 289, 324
statement number, finding in 294, 329
timestamp, finding in 298
variables, finding in 297, 330
ERROR ON-unit 280, 294, 316, 328
ers 279, 282, 315, 318
floating-point register 280, 316
object code listing 286
ON statement control block 286
static storage listing 285
SUBSCRIPTRANGE condition 281, 301, 304, 305, 316, 317, 334, 338
PLIDUMP subroutine 289, 325
PMR (problem management record) 525
pointer
variable 167, 475
PPA 523
preventive service planning (PSP) bucket 522
printf() function 25, 168, 476
problem management record (PMR) 525
procedure division listings 241
process
control block 62, 394
member list 62, 394
process control block (PCB) 62, 394
PROFILE run-time option
function 9, 360
program
class storage 246
program prolog area 523
program status word (PSW) 60, 393
PSP (preventive service planning) bucket 522
Q
QUIET suboption of TERMTHDACT run-time option 39, 375
R
reason code
nonzero returned to CICS 353
under CICS 349
registers 0–15
in dump 57
release number
in dump 56, 390
request parameter list (RPL) 198
return code
bad or nonzero 30, 372
RPTOPTS run-time option 9
RPTSTG run-time option 11, 361
run unit
COBOL 248
level control block 248
storage in dump 62, 248
run-time messages
under CICS 347
run-time options 20, 367
determining those in effect 9, 11, 360
sample options report 9
specifying 23
S
scope
terminator 237
SDUMP routine
description 268
format specifications 268
output 268
usage considerations 268
service routines
CDUMP/CPDUMP 267
DUMP/PDUMP 266
SDUMP 268
SET statement 238
shortcut keys 529
signal information in dump 197
sorted cross-reference listing 241
SOURCE compiler option 4, 5, 7
stack
frame 63
frame format 64, 65
STACK64 run-time option 361
STACKFRAME option of CEE3DMP callable service 38
statement numbers
  in dump  57, 390
static
  variables in dump  297, 330
  writable map  179, 180, 184
status
  of routines in dump  392
stderr  25, 372, 375, 376
stdio.h  168, 476
stdout  25
storage
  evaluating use of  11, 361
  for active routines  61, 393
  leak detecting  62, 112, 394, 428
  report  11, 361
  statistics  17, 18
STORAGE compiler option  7
storage dump
  written to an HFS file  224, 514
STORAGE option of CEE3DMP callable service  37
STORAGE run-time option  9, 360
structure
  map  183
  variable example code  183
symbolic
  feedback code  25, 368
symbolic dumps  268
  how to call under Fortran  268
syntax diagrams
  how to read  xix
system abend
  with TRAP(OFF)  30, 372
  with TRAP(ON)  30, 372
system dump
  generating  81, 394
  in z/OS UNIX shell  83

T

  task global table (TGT)  245
TERMINAL compiler option  4, 7
TERMTHDACT run-time option
  function  8, 39, 305, 338, 359, 375
  generating a dump and  21, 367
  modifying condition handling behavior and  9, 360
  suboptions  39, 375
TEST compiler option  4, 5, 6
TEST run-time option  9, 360
text file name prefixes, Language Environment  29, 371
THDCOM in dump  250
THREAD option of CEE3DMP callable service  37
THREADSTACK64 run-time option  361
time
  in dump  56, 390
TRACE run-time option
  function  9, 360
  trace table  154, 466
TRACE suboption of TERMTHDACT run-time option  39, 375
TRACEBACK option of CEE3DMP callable service  37, 62
transaction
  dump  349
  rollback  353
  rollback effects of assembler user exit on  353
  work area  349
TRAP run-time option
  function  9, 360
  Language Environment condition handling and  22, 81, 395
  Language Environment exception handling and  368

U

  UADUMP suboption of TERMTHDACT run-time option  21, 40, 367, 376
  UAIMM suboption of TERMTHDACT run-time option  21, 40, 367, 376
  UAONLY suboption of TERMTHDACT run-time option  21, 39, 367, 376
  UATRACE suboption of TERMTHDACT run-time option  21, 39, 367, 376
  unhandled conditions  24, 27, 370
    establishing enclave termination behavior for  24
  USE EXCEPTION/ERROR declaratives  238
  USE FOR DEBUGGING declarative  239
user
  abend  32, 373, 374
    code  30, 372
  exit  23, 24
  heap
    statistics  18
    stack
      statistics  17
  user-specified abends  32, 373
  USRHDLR run-time option  8, 22, 359
utility and service subroutines
  CDUMP/CPDUMP  267
  DUMP/PDUMP  266
  SDUMP  268

V

  variables
    in Language Environment dump  62
      structure example code  183
  VARIABLES option of CEE3DMP callable service  37, 62
  VBREF compiler option  5
  verb cross-reference  241
  Verbexit
    LEDATA  85, 397
  version number
    in dump  56, 390

W

  working storage
    in dump  61, 246, 393
  Writable Static Area  177
X

XPLINK
  downward-growing stack  64
  finding XPLINK information in a dump  206
  stack frame format  64, 65
  storage statistics  17
  trace table entries for  207
XREF compiler option  5, 7
XUFLOW run-time option
  function  9
  modifying condition handling behavior and  22

Z

z/OS Basic Skills information center  xxi
z/OS UNIX System Services
  C application program and  224, 514
  generating a system dump  83
Readers’ Comments — We’d Like to Hear from You

z/OS
Language Environment
Debugging Guide

Publication No. GA22-7560-10

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

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 state on this form.

Comments:

Thank you for your support.
Submit your comments using one of these channels:
• Send your comments to the address on the reverse side of this form.
• Send your comments via e-mail to: mhvrcfs@us.ibm.com

If you would like a response from IBM, please fill in the following information:

Name
Address

Company or Organization

Phone No. E-mail address