Contents

List of Figures ........................................................................................................................................ xi
List of Tables ..................................................................................................................................... xix

About this document .......................................................................................................................... xxii
Using your documentation ........................................................................................................... xxii
How to read syntax diagrams .................................................................................................... xxii
Symbols .......................................................................................................................................... xxii
Syntax items .................................................................................................................................... xxiii
Syntax examples .......................................................................................................................... xxiii
z/OS information ............................................................................................................................ xxiv

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

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

What Language Environment supports ........................................................................................... xxviii

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

Chapter 1. Preparing your routine for debugging ........................................................................ 3
Setting compiler options ............................................................................................................... 3
XL C and XL C++ compiler options ............................................................................................... 3
COBOL compiler options ............................................................................................................ 5
Fortran compiler options ............................................................................................................. 6
PL/I compiler options .................................................................................................................. 7
Enterprise PL/I for z/OS compiler options ................................................................................... 8
Using Language Environment runtime options ......................................................................... 8
Determining the runtime options in effect ................................................................................... 9
Understanding the HEAPZONES and HEAPCHK runtime options ......................................... 11
Using the CLER CICS transaction to display and set runtime options ................................... 11
Controlling storage allocation ................................................................................................... 11
Stack storage statistics ............................................................................................................. 16
Heap storage statistics .............................................................................................................. 18
HEAPPOOLS storage statistics ............................................................................................... 19
Modifying condition handling behavior .................................................................................. 20
Language Environment callable services .................................................................................. 20
Language Environment runtime options .................................................................................. 20
Customizing condition handlers ............................................................................................... 22
Invoking the assembler user exit ............................................................................................... 22
Establishing enclave termination behavior for unhandled conditions ................................... 23
Using messages in your routine ................................................................................................. 24
C/C++ ......................................................................................................................................... 24
COBOL ....................................................................................................................................... 24
Fortran ....................................................................................................................................... 24
PL/I ........................................................................................................................................... 24
Using condition information ..................................................................................................... 25
Using the feedback code parameter ......................................................................................... 25
Part 2. Debugging language-specific routines

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

Chapter 3. Using Language Environment debugging facilities
Debug tools
Language Environment dump service, CEE3DMP
Generating a Language Environment dump with CEE3DMP
Generating a Language Environment dump with TERMTHDACT
Generating a Language Environment dump with language-specific functions
Understanding the Language Environment dump
Debugging with specific sections of the Language Environment dump
Multiple enclave dumps
Generating a system dump
Steps for generating a system dump in a batch runtime environment
Steps for generating a system dump in an IMS runtime environment
Steps for generating a system dump in a CICS runtime environment
Steps for generating a Language Environment U4039 abend
Steps for generating a system dump in a z/OS UNIX shell
Formatting and analyzing system dumps
Preparing to use the Language Environment support for IPCS
Understanding Language Environment IPCS VERBEXIT – LEDATA
Understanding the Language Environment IPCS VERBEXIT LEDATA output
Understanding the HEAP LEDATA output
Understanding the heap pools LEDATA output
Understanding the heap pools trace LEDATA output
Understanding the C/C++-specific LEDATA output
Understanding the COBOL-specific LEDATA output
Understanding the PL/I-specific LEDATA output
Formatting individual control blocks
Controlling access to CEEDUMPS and DYNDUMPS
Requesting a Language Environment trace for debugging
Locating the trace dump
Using the Language Environment trace table format in a dump report
Understanding the trace table entry (TTE)
Sample dump for the trace table entry
Requesting a UNIX System Services syscall trace for debugging
<table>
<thead>
<tr>
<th>Chapter 4. Debugging C/C++ routines</th>
<th>163</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugging C/C++ programs..................................................</td>
<td>163</td>
</tr>
<tr>
<td>Using the __amrc and __amrc2 structures to debug input/output</td>
<td>163</td>
</tr>
<tr>
<td>Using file I/O tracing to debug C/C++ file I/O problems..........</td>
<td>169</td>
</tr>
<tr>
<td>Displaying an error message with the perror() function...........</td>
<td>169</td>
</tr>
<tr>
<td>Using __errno2() to diagnose application problems..................</td>
<td>169</td>
</tr>
<tr>
<td>Diagnosing DLL problems....................................................</td>
<td>171</td>
</tr>
<tr>
<td>Using C/C++ listings.........................................................</td>
<td>171</td>
</tr>
<tr>
<td>Finding variables.....................................................................</td>
<td>171</td>
</tr>
<tr>
<td>Generating a Language Environment dump of a C/C++ routine.........</td>
<td>178</td>
</tr>
<tr>
<td>cdump()</td>
<td>178</td>
</tr>
<tr>
<td>csnap()</td>
<td>179</td>
</tr>
<tr>
<td>ctrace()</td>
<td>179</td>
</tr>
<tr>
<td>Sample C routine that calls cdump().....................................</td>
<td>179</td>
</tr>
<tr>
<td>Sample C++ routine that generates a Language Environment dump...</td>
<td>180</td>
</tr>
<tr>
<td>Sample Language Environment dump with C/C++-specific information</td>
<td>181</td>
</tr>
<tr>
<td>Finding C/C++ information in a Language Environment dump........</td>
<td>186</td>
</tr>
<tr>
<td>Sample Language Environment dump with XPLINK-specific information</td>
<td>191</td>
</tr>
<tr>
<td>Finding XPLINK information in a Language Environment dump........</td>
<td>194</td>
</tr>
<tr>
<td>C/C++ contents of the Language Environment trace tables..........</td>
<td>195</td>
</tr>
<tr>
<td>Debugging examples of C/C++ routines...................................</td>
<td>203</td>
</tr>
<tr>
<td>Divide-by-zero error................................................................</td>
<td>203</td>
</tr>
<tr>
<td>Calling a nonexistent non-XPLINK function............................</td>
<td>207</td>
</tr>
<tr>
<td>Calling a nonexistent XPLINK function..................................</td>
<td>210</td>
</tr>
<tr>
<td>Handling dumps written to the z/OS UNIX file system................</td>
<td>213</td>
</tr>
<tr>
<td>Multithreading consideration.............................................</td>
<td>214</td>
</tr>
<tr>
<td>Understanding C/C++ heap information in storage reports..........</td>
<td>214</td>
</tr>
<tr>
<td>Language Environment storage report with heap pools statistics..</td>
<td>214</td>
</tr>
<tr>
<td>C function __uheapreport() storage report.............................</td>
<td>216</td>
</tr>
<tr>
<td>MEMCHECK VHM memory leak analysis tool.................................</td>
<td>218</td>
</tr>
<tr>
<td>Invoking MEMCHECK VHM.....................................................</td>
<td>218</td>
</tr>
<tr>
<td>MEMCHECK VHM environment variables.....................................</td>
<td>218</td>
</tr>
<tr>
<td>MEMCHECK VHM report sample scenario...................................</td>
<td>219</td>
</tr>
<tr>
<td>MEMCHECK VHM report examples...........................................</td>
<td>220</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5. Debugging COBOL programs</th>
<th>223</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determining the source of error........................................</td>
<td>223</td>
</tr>
<tr>
<td>Tracing program logic.....................................................</td>
<td>223</td>
</tr>
<tr>
<td>Finding input/output errors..............................................</td>
<td>223</td>
</tr>
<tr>
<td>Handling input/output errors............................................</td>
<td>224</td>
</tr>
<tr>
<td>Validating data (class test).............................................</td>
<td>224</td>
</tr>
<tr>
<td>Assessing switch problems................................................</td>
<td>224</td>
</tr>
<tr>
<td>Generating information about procedures.............................</td>
<td>224</td>
</tr>
<tr>
<td>Using COBOL listings.......................................................</td>
<td>225</td>
</tr>
<tr>
<td>Generating a Language Environment dump of a COBOL program.......</td>
<td>226</td>
</tr>
<tr>
<td>COBOL program that calls another COBOL program.....................</td>
<td>226</td>
</tr>
<tr>
<td>COBOL program that calls the Language Environment CEE3DMP callable service</td>
<td>227</td>
</tr>
<tr>
<td>Sample Language Environment dump with COBOL-specific information</td>
<td>227</td>
</tr>
<tr>
<td>Finding COBOL information in a dump...................................</td>
<td>228</td>
</tr>
<tr>
<td>Debugging example COBOL programs.....................................</td>
<td>233</td>
</tr>
<tr>
<td>Subscript range error......................................................</td>
<td>233</td>
</tr>
<tr>
<td>Calling a nonexistent subroutine.......................................</td>
<td>235</td>
</tr>
<tr>
<td>Divide-by-zero error.......................................................</td>
<td>239</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6. Debugging Fortran routines</th>
<th>245</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determining the source of errors in Fortran routines...............</td>
<td>245</td>
</tr>
<tr>
<td>Identifying runtime errors...............................................</td>
<td>245</td>
</tr>
</tbody>
</table>
Chapter 8. Debugging Enterprise PL/I routines......................................................................................287
  Determining the source of errors in Enterprise PL/I routines..........................................................287
    Logic errors in the source routine.................................................................................................287
    Invalid use of Enterprise PL/I......................................................................................................287
    Unforeseen errors.........................................................................................................................288
    Invalid input data ........................................................................................................................288
    Compiler or runtime routine malfunction......................................................................................288
    System malfunction.......................................................................................................................288
    Unidentified routine malfunction................................................................................................288
    Storage overlay problems.............................................................................................................289
  Using Enterprise PL/I compiler listings............................................................................................290
    Generating Enterprise PL/I listings and maps...............................................................................290
    Finding information in Enterprise PL/I listings............................................................................290
  Generating a Language Environment dump of an Enterprise PL/I routine........................................296
    PLIDUMP syntax and options.......................................................................................................296
    PLIDUMP usage notes...................................................................................................................298
  Finding Enterprise PL/I information in a dump................................................................................298
    Traceback....................................................................................................................................299
    Control blocks for active routines...............................................................................................301
    Control blocks associated with the thread....................................................................................301
  Enterprise PL/I contents of the Language Environment trace table................................................302
  Debugging example of Enterprise PL/I routines...............................................................................303
  Invalid use of Enterprise PL/I........................................................................................................293
  Invalid use of PL/I for MVS & VM.................................................................................................261

Chapter 7. Debugging PL/I for MVS & VM routines.................................................................................259
  Determining the source of errors in PL/I for MVS & VM routines.....................................................259
    Logic errors in the source routine.................................................................................................259
    Invalid use of PL/I for MVS & VM...............................................................................................259
    Unforeseen errors........................................................................................................................260
    Invalid input data........................................................................................................................260
    Compiler or runtime routine malfunction......................................................................................260
    System malfunction.......................................................................................................................260
    Unidentified routine malfunction................................................................................................260
    Storage overlay problems.............................................................................................................261
  Using PL/I for MVS & VM compiler listings....................................................................................262
    Generating PL/I for MVS & VM listings and maps.......................................................................262
    Finding information in PL/I for MVS & VM listings.....................................................................262
  Generating a Language Environment dump of a PL/I for MVS & VM routine....................................268
    PLIDUMP syntax and options.......................................................................................................268
    PLIDUMP usage notes...................................................................................................................270
  Finding PL/I for MVS & VM information in a dump..........................................................................270
    Traceback....................................................................................................................................271
    Control blocks for active routines...............................................................................................273
    Control blocks associated with the thread....................................................................................275
  PL/I for MVS & VM contents of the Language Environment trace table..........................................277
  Debugging example of PL/I for MVS & VM routines........................................................................277
    Subscript range error....................................................................................................................277
    Calling a nonexistent subroutine..................................................................................................281
    Divide-by-zero error......................................................................................................................283

Examples of debugging Fortran routines..............................................................................................254
  Calling a nonexistent routine...........................................................................................................254
  Divide-by-zero error.........................................................................................................................255
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscript range error</td>
<td>303</td>
</tr>
<tr>
<td>Calling a nonexistent subroutine</td>
<td>306</td>
</tr>
<tr>
<td>Divide-by-zero error</td>
<td>307</td>
</tr>
<tr>
<td>Chapter 9. Debugging under CICS</td>
<td>313</td>
</tr>
<tr>
<td>Accessing debugging information</td>
<td>313</td>
</tr>
<tr>
<td>Locating Language Environment runtime messages</td>
<td>313</td>
</tr>
<tr>
<td>Locating the Language Environment traceback</td>
<td>313</td>
</tr>
<tr>
<td>Locating the Language Environment dump</td>
<td>314</td>
</tr>
<tr>
<td>Using CICS transaction dump</td>
<td>314</td>
</tr>
<tr>
<td>Using CICS register and program status word contents</td>
<td>314</td>
</tr>
<tr>
<td>Using Language Environment abend and reason codes</td>
<td>315</td>
</tr>
<tr>
<td>Using Language Environment return codes to CICS</td>
<td>315</td>
</tr>
<tr>
<td>Activating Language Environment feature trace records under CICS</td>
<td>316</td>
</tr>
<tr>
<td>Ensuring transaction rollback</td>
<td>317</td>
</tr>
<tr>
<td>Finding data when Language Environment returns a nonzero return code</td>
<td>317</td>
</tr>
<tr>
<td>Finding data when Language Environment abends internally</td>
<td>318</td>
</tr>
<tr>
<td>Finding data when Language Environment abends from an EXEC CICS command</td>
<td>318</td>
</tr>
<tr>
<td>Displaying and modifying runtime options with the CLER transaction</td>
<td>319</td>
</tr>
</tbody>
</table>

**Part 3. Debugging Language Environment AMODE 64 applications** 321

| Chapter 10. Preparing your AMODE 64 application for debugging          | 323  |
| Setting compiler options                                              | 323  |
| XL C and XL C++ compiler options for AMODE 64 applications             | 323  |
| Using Language Environment runtime options                             | 323  |
| Determining the runtime options in effect for AMODE 64 applications    | 324  |
| Understanding the HEAPZONES and HEAPCHK runtime options               | 325  |
| Controlling storage allocation for AMODE 64 applications               | 326  |
| Storage statistics for AMODE 64 applications                           | 328  |
| Modifying exception handling behavior                                  | 329  |
| Language Environment application program interfaces (API)              | 329  |
| Language Environment runtime options                                   | 330  |
| Customizing exception handlers                                         | 330  |
| Using condition information                                            | 330  |
| Using the feedback code parameter                                      | 331  |
| Using the symbolic feedback code                                       | 332  |

| Chapter 11. Classifying AMODE 64 application errors                    | 333  |
| Identifying problems in routines                                      | 333  |
| Language Environment module names                                      | 333  |
| Common errors in routines                                              | 333  |
| Interpreting runtime messages                                          | 334  |
| Message prefix                                                         | 335  |
| Message number                                                         | 335  |
| Severity code                                                         | 335  |
| Message text                                                          | 335  |
| Understanding abend codes                                              | 335  |
| User abends                                                           | 336  |
| System abends                                                         | 336  |

| Chapter 12. Using Language Environment AMODE 64 debugging facilities  | 337  |
| Debugging tools                                                       | 337  |
| Language Environment dumps                                            | 337  |
| Generating a Language Environment dump with TERMTHDACT                 | 337  |
| Generating a Language Environment dump with language-specific functions| 339  |
| Understanding the Language Environment dump                           | 339  |
Generating a system dump.................................................................354
Steps for generating a system dump in a batch runtime environment.................................355
Steps for generating a system dump in a z/OS UNIX shell..............................................................355
Formatting and analyzing system dumps......................................................................................356
Preparing to use the Language Environment support for IPCS................................................356
Understanding Language Environment IPCS VERBEXIT – LEDATA........................................356
Understanding the Language Environment IPCS VERBEXIT LEDATA output..........................360
Understanding the HEAP LEDATA output.....................................................................................375
Understanding the heap pool LEDATA output...............................................................................384
Understanding the heap pools trace LEDATA output.....................................................................390
Understanding the C/C++-specific LEDATA output........................................................................394
C/C++-specific sections of the LEDATA output..............................................................................404
Understanding the PL/I-specific LEDATA output...........................................................................406
PL/I-specific sections of the LEDATA output...................................................................................411
Understanding the AUTH LEDATA output........................................................................................411
Sections of the AUTH LEDATA VERBEXIT formatted output......................................................415
Formatting individual control blocks.............................................................................................417
Requesting a Language Environment trace for debugging............................................................419
Locating the trace dump..................................................................................................................420
Using the Language Environment trace table format in a dump report.........................................420
Understanding the trace table entry (TTE)....................................................................................420
Sample dump for the trace table entry............................................................................................428
Requesting a UNIX System Services syscall trace for debugging....................................................428

Chapter 13. Debugging AMODE 64 C/C++ routines........................................................................429
Debugging C/C++ programs.............................................................................................................429
Using the __amrc and __amrc2 structures to debug input/output..................................................429
Using file I/O tracing to debug C/C++ file I/O problems.................................................................435
Displaying an error message with the perror() function...............................................................435
Using __errno2() to diagnose application problems.......................................................................436
Using C/C++ listings.......................................................................................................................437
Finding variables.............................................................................................................................437
Generating a Language Environment dump of a C/C++ routine....................................................440
cdump()............................................................................................................................................440
csnap().............................................................................................................................................441
ctrace()............................................................................................................................................441
Sample C routine that calls cdump()...............................................................................................441
Sample C++ routine that generates a Language Environment dump.............................................443
Sample Language Environment dump with C/C++-specific information........................................444
C/C++ contents of the Language Environment trace tables............................................................448
Debugging examples of C/C++ routines............................................................................................451
Divide-by-zero error.........................................................................................................................451
Calling a nonexistent function..........................................................................................................457
Handling dumps written to the z/OS UNIX file system....................................................................462
Multithreading consideration.............................................................................................................463
Understanding C/C++ heap information in storage reports..............................................................463
Language Environment storage report with heap pools statistics.................................................463
C function __uheapreport() storage report......................................................................................466

Appendix A. Diagnosing problems with Language Environment..............................................469
Diagnosis checklist............................................................................................................................469
Locating the name of the failing routine for a non-XPLINK application.........................................470
Searching the IBM Software Support Database..............................................................................472
Preparing documentation for an authorized program analysis report (APAR).................................473

Appendix B. Accessibility.................................................................477
Accessibility features.........................................................................................................................477
Consult assistive technologies................................................................................................................ 477
Keyboard navigation of the user interface.............................................................................................. 477
Dotted decimal syntax diagrams.............................................................................................................477

Notices..................................................................................................................................................... 481
Terms and conditions for product documentation................................................................................. 482
IBM Online Privacy Statement.................................................................................................................. 483
Policy for unsupported hardware.............................................................................................................. 483
Minimum supported hardware.................................................................................................................. 484
Programming interface information.......................................................................................................... 484
Trademarks.............................................................................................................................................. 484

Index....................................................................................................................................................... 485
# List of Figures

1. Options report example produced by runtime option RPTOPTS(ON) ........................................................ 10
2. Storage report produced by runtime option RPTSTG(ON) ........................................................................ 12
3. Storage report produced by runtime option RPTSTG(ON) (continued) ..................................................... 13
4. Storage report produced by RPTSTG(ON) with XPLINK ............................................................................. 14
5. Storage report produced by RPTSTG(ON) with XPLINK (continued) ......................................................... 15
6. Storage report produced by RPTSTG(ON) with XPLINK (continued) ......................................................... 16
7. Language Environment condition token ..................................................................................................... 26
8. The C program CELSAMP ............................................................................................................................ 41
9. The C program CELSAMP (continued) ........................................................................................................ 42
10. The C program CELSAMP (continued) ...................................................................................................... 43
11. The C DLL CELDLL ..................................................................................................................................... 44
12. Upward-growing (non-XPLINK) stack frame format ................................................................................ 63
13. Downward-growing (XPLINK) stack frame format ................................................................................... 64
14. Condition information block (Part A) ........................................................................................................ 76
15. Condition information block (Part B) ........................................................................................................ 77
16. Machine state information block .............................................................................................................. 79
17. Language Environment dump of multiple enclaves ................................................................................ 80
18. Example of formatted output from LEDATA VERBEXIT (Part 1 of 16) ..................................................... 89
19. Example of formatted output from LEDATA VERBEXIT (Part 2 of 16) ..................................................... 89
20. Example of formatted output from LEDATA VERBEXIT (Part 3 of 16) ..................................................... 90
21. Example of formatted output from LEDATA VERBEXIT (Part 4 of 16) ..................................................... 91
22. Example of formatted output from LEDATA VERBEXIT (Part 5 of 16) ..................................................... 92
23. Example of formatted output from LEDATA VERBEXIT (Part 6 of 16) ..................................................... 93
24. Example of formatted output from LEDATA VERBEXIT (Part 7 of 16) ..................................................... 94
25. Example of formatted output from LEDATA VERBEXIT (Part 8 of 16) ..................................................... 95
26. Example of formatted output from LEDATA VERBEXIT (Part 9 of 16) ..................................................... 96
27. Example of formatted output from LEDATA VERBEXIT (Part 10 of 16) ................................................... 97
28. Example of formatted output from LEDATA VERBEXIT (Part 11 of 16) ................................................ 98
29. Example of formatted output from LEDATA VERBEXIT (Part 12 of 16) ................................................ 99
30. Example of formatted output from LEDATA VERBEXIT (Part 13 of 16) ................................................ 100
31. Example of formatted output from LEDATA VERBEXIT (Part 14 of 16) ............................................... 101
32. Example of formatted output from LEDATA VERBEXIT (Part 15 of 16) ................................................. 102
33. Example of formatted output from LEDATA VERBEXIT (Part 16 of 16) ................................................ 103
34. CAA formatted by the CBFORMAT IPCS command ............................................................................... 148
35. Format of the trace table entry ............................................................................................................... 152
36. __amrc structure ..................................................................................................................................... 164
37. __amrc2 structure ................................................................................................................................... 164
38. Example of a routine using perror() ........................................................................................................ 169
39. Example of a routine using __errno2() ..................................................................................................... 170
<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>How to use z/OS Language Environment publications</td>
<td>xxii</td>
</tr>
<tr>
<td>2</td>
<td>Syntax examples</td>
<td>xxiii</td>
</tr>
<tr>
<td>3</td>
<td>TEST suboptions to simplify debugging</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>C/C++ compiler options to simplify runtime debugging</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>COBOL compiler options for runtime debugging</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Fortran compiler options for runtime debugging</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>PL/I compiler options for debugging</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>Enterprise PL/I for z/OS compiler options for debugging</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>Language Environment runtime options for debugging</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>Storage report fields that display stack storage statistics</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>Storage report fields that display heap storage statistics</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>Callable services that modify condition handling</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>Runtime options that modify condition handling</td>
<td>21</td>
</tr>
<tr>
<td>14</td>
<td>Callable services that can convert condition tokens to routine variables, messages, or signaled conditions</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>Common error symptoms, possible causes, and programmer responses</td>
<td>28</td>
</tr>
<tr>
<td>16</td>
<td>CEE3DMP options</td>
<td>34</td>
</tr>
<tr>
<td>17</td>
<td>TERMTHDACT suboptions, level of information, and destinations</td>
<td>36</td>
</tr>
<tr>
<td>18</td>
<td>Condition handling of 0Cx abends</td>
<td>39</td>
</tr>
<tr>
<td>19</td>
<td>Contents of the Language Environment dump</td>
<td>55</td>
</tr>
<tr>
<td>20</td>
<td>Description of CAA fields</td>
<td>65</td>
</tr>
<tr>
<td>21</td>
<td>Contents of the LEDATA VERBEXIT formatted output</td>
<td>103</td>
</tr>
<tr>
<td>22</td>
<td>Contents of the Heap report sections of LEDATA output</td>
<td>110</td>
</tr>
<tr>
<td>23</td>
<td>Contents of heap pools report sections of LEDATA output</td>
<td>115</td>
</tr>
<tr>
<td>24</td>
<td>Contents of heap pools trace section of LEDATA output</td>
<td>117</td>
</tr>
<tr>
<td>25</td>
<td>Contents of C/C++-specific sections of LEDATA output</td>
<td>136</td>
</tr>
<tr>
<td>26</td>
<td>Contents of COBOL-specific sections of LEDATA Output</td>
<td>139</td>
</tr>
<tr>
<td>27</td>
<td>Contents of COBOL-specific sections of LEDATA Output (Enterprise COBOL V5.1 and later releases)</td>
<td>141</td>
</tr>
<tr>
<td>28</td>
<td>Contents of PL/I-specific sections of LEDATA output</td>
<td>147</td>
</tr>
<tr>
<td>29</td>
<td>Language Environment Control blocks that can be individually formatted</td>
<td>148</td>
</tr>
<tr>
<td>30</td>
<td>TERMTHDACT runtime option settings and dump contents produced</td>
<td>150</td>
</tr>
<tr>
<td>31</td>
<td>LE=1 entry records</td>
<td>153</td>
</tr>
<tr>
<td>32</td>
<td>LE=2 entry records</td>
<td>153</td>
</tr>
<tr>
<td>33</td>
<td>Format of the mutex/CV/latch records</td>
<td>158</td>
</tr>
<tr>
<td>34</td>
<td>LE=8 entry records</td>
<td>159</td>
</tr>
<tr>
<td>35</td>
<td>LE=20 entry records</td>
<td>159</td>
</tr>
<tr>
<td>36</td>
<td>__last_op values and diagnosis information</td>
<td>166</td>
</tr>
<tr>
<td>37</td>
<td>Finding the WSA base address</td>
<td>172</td>
</tr>
</tbody>
</table>
38. Contents of the COBOL sections of Language Environment..........................................................186
39. Contents of XPLINK information in a Language Environment dump.....................................................195
40. Compiler-generated COBOL listings and their contents........................................................................226
41. Compiler-generated Fortran listings and their contents.........................................................................247
42. Understanding the Language Environment traceback table...................................................................253
43. Compiler-generated PL/I for MVS & VM listings and their contents.......................................................262
44. Typical comments in a PL/I for MVS & VM static storage listing............................................................265
45. Comments in a PL/I for MVS & VM object code listing...........................................................................266
46. PL/I for MVS & VM mnemonics...............................................................................................................267
47. Compiler-generated PL/I listings and their contents............................................................................290
48. Finding data when Language Environment returns a nonzero return code...........................................318
49. Finding data when Language Environment abends internally...............................................................318
50. Finding data when Language Environment abends from an EXEC CICS command.............................318
51. Common error symptoms, possible causes, and programmer responses................................................334
52. TERMTHDACT suboptions, level of information, and destinations.......................................................337
53. Contents of the Language Environment dump - AMODE 64..................................................................350
54. Contents of the Language Environment LEDATA VERBEXIT formatted output (AMODE 64).............371
55. Contents of Heap report sections of the LEDATA output (AMODE 64)..................................................383
56. Contents of the heap pool report sections of the LEDATA output (AMODE 64)........................................390
57. Contents of a detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64)..........................394
58. Contents of C/C++-specific sections of the LEDATA output (AMODE 64).................................................404
59. Contents of PL/I-specific sections of the LEDATA output (AMODE 64)....................................................411
60. Contents of AUTH LEDATA VERBEXIT formatted output (AMODE 64).............................................416
61. Language Environment control blocks that can be individually formatted............................................418
62. Preinitalized Environments for Authorized Programs control blocks that can be individually formatted..........................................................................................418
63. TERMTHDACT runtime option settings and dump contents produced (AMODE 64)...............................419
64. LE=1 entry records (AMODE 64).............................................................................................................422
65. LE=2 entry records (AMODE 64).............................................................................................................422
66. Format of the mutex/CV/latch records (AMODE 64).................................................................................427
67. LE=8 entry records (AMODE 64).............................................................................................................427
68. __last_op values and diagnosis information (AMODE 64)......................................................................432
69. Problem resolution documentation requirements..................................................................................474
About this document

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

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

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

Using your documentation

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

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

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

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

<table>
<thead>
<tr>
<th>To ...</th>
<th>Use ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluate Language Environment</td>
<td><em>z/OS Language Environment Concepts Guide</em></td>
</tr>
</tbody>
</table>
| Plan for Language Environment | *z/OS Language Environment Concepts Guide*  
| | *z/OS Language Environment Runtime Application Migration Guide* |
| Install Language Environment | *z/OS Program Directory* in the *z/OS Internet library*  
<p>| Customize Language Environment | <em>z/OS Language Environment Customization</em> |</p>
<table>
<thead>
<tr>
<th>To ...</th>
<th>Use ...</th>
</tr>
</thead>
</table>
| **Understand Language Environment program models and concepts** | z/OS Language Environment Concepts Guide  
z/OS Language Environment Programming Guide  
z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode |
| **Find syntax for Language Environment runtime options and callable services** | z/OS Language Environment Programming Reference |
| **Develop applications that run with Language Environment** | z/OS Language Environment Programming Guide and your language programming guide |
| **Debug applications that run with Language Environment, diagnose problems with Language Environment** | z/OS Language Environment Debugging Guide |
| **Get details on runtime messages** | z/OS Language Environment Runtime Messages |
| **Develop interlanguage communication (ILC) applications** | z/OS Language Environment Writing Interlanguage Communication Applications and your language programming guide |
| **Migrate applications to Language Environment** | z/OS Language Environment Runtime Application Migration Guide and the migration guide for each Language Environment-enabled language |

### How to read syntax diagrams

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

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

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

### Symbols

The following symbols may be displayed in syntax diagrams:

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

Syntax diagrams contain many different items. Syntax items include:

- Keywords - a command name or any other literal information.
- Variables - variables are italicized, appear in lowercase, and represent the name of values you can supply.
- Delimiters - delimiters indicate the start or end of keywords, variables, or operators. For example, a left parenthesis is a delimiter.
- Operators - operators include add (+), subtract (-), multiply (*), divide (/), equal (=), and other mathematical operations that may need to be performed.
- Fragment references - a part of a syntax diagram, separated from the diagram to show greater detail.
- Separators - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.

**Note:** If a syntax diagram shows a character that is not alphanumeric (for example, parentheses, periods, commas, equal signs, a blank space), enter the character as part of the syntax.

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

**Item type**

**Definition**

**Required**
- Required items are displayed on the main path of the horizontal line.

**Optional**
- Optional items are displayed below the main path of the horizontal line.

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

Syntax examples

The following table provides syntax examples.

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required item.</td>
<td>Required items appear on the main path of the horizontal line. You must specify these items.</td>
</tr>
<tr>
<td>Required choice.</td>
<td>A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.</td>
</tr>
<tr>
<td>Optional item.</td>
<td>Optional items appear below the main path of the horizontal line.</td>
</tr>
<tr>
<td>Optional choice.</td>
<td>An optional choice (two or more items) appears in a vertical stack below the main path of the horizontal line. You may choose one of the items in the stack.</td>
</tr>
</tbody>
</table>
Table 2: Syntax examples. (continued)

<table>
<thead>
<tr>
<th>Item</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default.</td>
<td>Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.</td>
</tr>
<tr>
<td>Variable.</td>
<td>Variables appear in lowercase italics. They represent names or values.</td>
</tr>
<tr>
<td>Repeatable item.</td>
<td>An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated. A character within the arrow means you must separate repeated items with that character. An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.</td>
</tr>
<tr>
<td>Fragment.</td>
<td>The fragment symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.</td>
</tr>
</tbody>
</table>

**z/OS information**

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

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

To find the complete z/OS® library, go to IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSLTBW/welcome).
How to send your comments to IBM

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

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

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

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

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

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

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

• Your name, company/university/institution name, and email address
• The following deliverable title and order number: z/OS Language Environment Debugging Guide, GA32-0908-30
• The section title of the specific information to which your comment relates
• The text of your comment.

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

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

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

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

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

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

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

New

Because APAR PI91583 added support for lengths that are longer than 7 bytes, the service portion of Traceback for both CEEDUMP and LEDATA Verbexit was updated. See “Sections of the Language Environment dump” on page 54 and “Sections of the Language Environment LEDATA VERBEXIT formatted output” on page 103.

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

- Chapter 3, “Using Language Environment debugging facilities,” on page 33
  - “Understanding the C/C++-specific LEDATA output” on page 118
- Chapter 12, “Using Language Environment AMODE 64 debugging facilities,” on page 337
  - “Understanding the C/C++-specific LEDATA output” on page 394

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

- “Common Anchor Area” on page 64

Language Environment now enforces the same security rules that are enforced by ABEND dump processing. For more information, see the new section “Controlling access to CEEDUMPs and DYNDUMPs” on page 149.

Support was added to allow vector applications to run under ERTLI CICS. The following section contains new information:

- “Common Anchor Area” on page 64

Changed

- These report headers were updated for V2R3:
  - The options report header (Figure 1 on page 10)
  - The storage report header (Figure 2 on page 12)
  - The options report for AMODE 64 applications (Figure 156 on page 325)
  - The storage report for AMODE 64 applications (Figure 157 on page 326)
  - “MEMCHECK VHM memory leak analysis tool” on page 218 now indicates that the trace is limited to 1024 entries and will wrap.
  - Various updates were made to replace IBM Debug Tool for z/OS with IBM z/OS Debugger.
Summary of changes for Language Environment for z/OS Version 2 Release 2 (V2R2)

**Changed**

The following report headers were updated for V2R2:

- The options report header (Figure 1 on page 10)
- The storage report header (Figure 2 on page 12)
- The options report for AMODE 64 applications (Figure 156 on page 325)
- The storage report for AMODE 64 applications (Figure 157 on page 326)

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

**New**

- Support was added for vectors. The following topics contain new information for this support:
  - “Understanding the Language Environment dump” on page 40
  - “Vector registers” on page 189
  - Chapter 5, “Debugging COBOL programs,” on page 223
  - “Divide-by-zero error” on page 307
  - “Understanding the Language Environment IPCS VERBEXIT LEDATA output” on page 360
  - “Divide-by-zero error” on page 451

**z/OS Version 2 Release 1 summary of changes**

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

- z/OS Migration
- z/OS Planning for Installation
- z/OS Summary of Message and Interface Changes
- z/OS Introduction and Release Guide
What Language Environment supports

Language Environment supports z/OS (5650-ZOS).

IBM Language Environment (also called Language Environment) provides common services and language-specific routines in a single runtime environment for C, C++, COBOL, Fortran (z/OS only; no support for z/OS UNIX 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 runtime environment for applications that are generated with the following IBM compiler products:

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

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

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

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

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

For more information about VisualAge for Java, Enterprise Edition for OS/390, program number 5655-JAV, see the product documentation.
Part 1. Introduction to debugging in Language Environment

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

This chapter describes options and features that you can use to prepare your routine for debugging. The following topics are covered:

• Compiler options for C, C++, COBOL, Fortran, and PL/I
• Language Environment runtime options
• Use of storage in routines
• Options for modifying condition handling
• Assembler user exits
• Enclave termination behavior
• User-created messages
• Language Environment feedback codes and condition tokens

Setting compiler options

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

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

XL C and XL C++ compiler options

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

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

<table>
<thead>
<tr>
<th>Suboption name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYM</td>
<td>Generates symbol table information and enables Language Environment to generate a dump at run time. When you specify SYM, you also get the value and type of variables displayed in the Local Variables section of the dump. For example, if in block 4 the variable x is a signed integer of 12, and in block 2 the variable x is a signed integer of 1, the following output appears in the Local Variables section of the dump:</td>
</tr>
<tr>
<td>%BLOCK4:x</td>
<td>signed int</td>
</tr>
<tr>
<td>%BLOCK2:x</td>
<td>signed int</td>
</tr>
</tbody>
</table>

If a nonzero optimization level is used, variables do not appear in the dump.

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

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

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGGREGATE (C)</td>
<td>Specifies that a layout for struct and union type variables appear in the listing.</td>
</tr>
<tr>
<td>ATTRIBUTE (C++)</td>
<td>For XL C++ compile, cross reference listing with attribute information. If XREF is specified, the listing also contains reference, definition and modification information.</td>
</tr>
<tr>
<td>CHECKOUT (C)</td>
<td>Provides informational messages indicating possible programming errors.</td>
</tr>
<tr>
<td>EVENTS</td>
<td>Produces an events file that contains error information and source file statistics.</td>
</tr>
<tr>
<td>EXPMAC</td>
<td>Macro expansions with the original source.</td>
</tr>
<tr>
<td>FLAG</td>
<td>Specifies the minimum severity level that is tolerated.</td>
</tr>
<tr>
<td>GONUMBER</td>
<td>Generates line number tables corresponding to the input source file. This option is turned on when the TEST option is used. This option is needed to show statement numbers in dump output.</td>
</tr>
<tr>
<td>INFO (C++)</td>
<td>Indication of possible programming errors.</td>
</tr>
<tr>
<td>INLINE</td>
<td>Inline Summary and Detailed Call Structure Reports. (Specify with the REPORT sub-option).</td>
</tr>
<tr>
<td>INLRPT</td>
<td>Generates a report on status of functions that were inlined. The OPTIMIZE option must also be specified.</td>
</tr>
<tr>
<td>LIST</td>
<td>Listing of the pseudo-assembly listing produced by the compiler.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>Displays the offset addresses relative to the entry point of each function.</td>
</tr>
<tr>
<td>PHASEID</td>
<td>Causes each compiler module (phase) to issue an informational message which identifies the compiler phase module name, product identifier, and build level.</td>
</tr>
<tr>
<td>PPONLY</td>
<td>Completely expanded z/OS XL C, or z/OS XL C++ source code, by activating the preprocessor (PP) only. The output shows, for example, all the &quot;#include&quot; and &quot;#define&quot; directives.</td>
</tr>
<tr>
<td>SERVICE</td>
<td>Places a string in the object module, which is displayed in the traceback if the application fails abnormally.</td>
</tr>
<tr>
<td>SHOWINC</td>
<td>All included text in the listing.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Includes source input statements and diagnostic messages in the listing.</td>
</tr>
<tr>
<td>TERMINAL</td>
<td>Directs all error messages from the compiler to the terminal. If not specified, this is the default.</td>
</tr>
</tbody>
</table>
Table 4: C/C++ compiler options to simplify runtime debugging. (continued)

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>Generates information for debugging interface. This also generates symbol tables needed for symbolic variables in the dump.</td>
</tr>
<tr>
<td>XPLINK (BACKCHAIN)</td>
<td>Generates a prolog that saves redundant information in the calling function's stack frame.</td>
</tr>
<tr>
<td>XPLINK (STOREARGS)</td>
<td>Generates code to store arguments that are normally passed in registers, into the argument area.</td>
</tr>
<tr>
<td>XREF</td>
<td>For XL C compile, cross reference listing with reference, definition, and modification information. For XL C++ compile, cross reference listing with reference, definition, and modification information. If you specify ATTRIBUTE, the listing also contains attribute information.</td>
</tr>
</tbody>
</table>

See the Inter-procedural Analysis chapter in the z/OS XL C/C++ Programming Guide for an overview and more details about Inter-procedural Analysis.

COBOL compiler options

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

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

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

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

Table 5: COBOL compiler options for runtime debugging.

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST</td>
<td>Produces a listing of the assembler expansion of your source code and global tables, literal pools, information about working storage, and size of routine's working storage.</td>
</tr>
<tr>
<td>MAP</td>
<td>Produces lists of items in data division including a data division map, global tables, literal pools, a nested program structure map, and attributes.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>Produces a condensed PROCEDURE DIVISION listing containing line numbers, statement references, and location of the first instruction generated for each statement.</td>
</tr>
<tr>
<td>OUTDD</td>
<td>Specifies the destination of DISPLAY statement messages.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Produces a listing of your source program with any statements embedded by PROCESS, COPY, or BASIS statements.</td>
</tr>
<tr>
<td>TEST</td>
<td>Produces object code that can run with a debugging tool, or adds information to the object program to produce formatted dumps. With or without any suboptions, this option forces the OBJECT option. When specified with any of the hook-location suboption values except NONE, this option forces the NOOPTIMIZE option. DWARF suboption includes statement numbers in the Language Environment dump and produces a symbolic dump.</td>
</tr>
</tbody>
</table>

**Note:** For COBOL V4R2 and prior releases, use the SYM suboption instead of DWARF.
Table 5: COBOL compiler options for runtime debugging. (continued)

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBREF</td>
<td>Produces a cross-reference of all verb types used in the source program and a summary of how many times each verb is used.</td>
</tr>
<tr>
<td>XREF</td>
<td>Creates a sorted cross-reference listing.</td>
</tr>
</tbody>
</table>

Fortran compiler options

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

Table 6: Fortran compiler options for runtime debugging.

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIPS</td>
<td>Specifies if standard language flagging is to be performed. This is valuable if you want to write a program conforming to Fortran 77.</td>
</tr>
<tr>
<td>FLAG</td>
<td>Specifies the level of diagnostic messages to be written. I (Information), E (Error), W (Warning), or S (Severe). You can also use FLAG to suppress messages that are below a specified level. This is useful if you want to suppress information messages, for example.</td>
</tr>
<tr>
<td>GOSTMT</td>
<td>Specifies that statement numbers are included in the runtime messages and in the Language Environment dump.</td>
</tr>
<tr>
<td>ICA</td>
<td>Specifies if intercompilation analysis is to be performed, specifies the files containing intercompilation analysis information to be used or updated, and controls output from intercompilation analysis. Specify ICA when you have a group of programs and subprograms that you want to process together and you need to know if there are any conflicting external references, mismatched commons, and so on.</td>
</tr>
<tr>
<td>LIST</td>
<td>Specifies if the object module list is to be written. The LIST option lets you see the pseudo-assembly language code that is similar to what is actually generated.</td>
</tr>
<tr>
<td>MAP</td>
<td>Specifies if a table of source program variable names, named constants, and statement labels and their displacements is to be produced.</td>
</tr>
<tr>
<td>OPTIMIZE</td>
<td>Specifies the optimizing level to be used during compilation. If you are debugging your program, it is advisable to use NOOPTIMIZE.</td>
</tr>
<tr>
<td>SDUMP</td>
<td>Specifies if dump information is to be generated.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Specifies if a source listing is to be produced.</td>
</tr>
<tr>
<td>SRCFLG</td>
<td>Controls insertion of error messages in the source listing. SRCFLG allows you to view error messages after the initial line of each source statement that caused the error, rather than at the end of the listing.</td>
</tr>
<tr>
<td>SXM</td>
<td>Formats SREF or MAP listing output to a 72-character width.</td>
</tr>
<tr>
<td>SYM</td>
<td>Invokes the production of SYM cards in the object text file. SYM cards contain location information for variables within a Fortran program.</td>
</tr>
<tr>
<td>TERMINAL</td>
<td>Specifies whether error messages and compiler diagnostics are to be written on the SYSTERM data set and whether a summary of messages for all the compilations is to be written at the end of the listing.</td>
</tr>
<tr>
<td>TEST</td>
<td>Specifies whether to override any optimization level above OPTIMIZE(0). This option adds runtime overhead.</td>
</tr>
<tr>
<td>TRMFLG</td>
<td>Specifies whether to display the initial line of source statements in error and their associated error messages at the terminal.</td>
</tr>
<tr>
<td>XREF</td>
<td>Creates a cross-reference listing.</td>
</tr>
</tbody>
</table>
Table 6: Fortran compiler options for runtime debugging. (continued)

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VECTOR</td>
<td>Specifies whether to invoke the vectorization process. A vectorization report provides detailed information about the vectorization process.</td>
</tr>
</tbody>
</table>

PL/I compiler options

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

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

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

Table 7: PL/I compiler options for debugging.

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

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

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

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGGREGATE</td>
<td>Specifies that a layout for arrays and major structures appears in the listing.</td>
</tr>
<tr>
<td>GONUMBER / GOSTMT</td>
<td>Tells the compiler to produce additional information specifying that line numbers from the source routine can be included in runtime messages and in the Language Environment dump.</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>Specifies that users can establish an ATTENTION ON-unit that gains control when an attention interrupt occurs.</td>
</tr>
<tr>
<td>LIST</td>
<td>Produces a listing of the assembler expansion of source code and global tables, literal pools, information about working storage, and size of routine's working storage.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>Displays the offset addresses relative to the entry point of each function.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Specifies that the compiler includes a listing of the source routine in the listing.</td>
</tr>
<tr>
<td>STORAGE</td>
<td>Includes a table of the main storage requirements for the object module in the listing.</td>
</tr>
<tr>
<td>TEST</td>
<td>Specifies the level of testing capability that is generated as part of the object code. TEST also controls the location of test hooks and whether or not the symbol table is generated. Because the TEST option increases the size of the object code and can affect performance, limit the number and placement of hooks.</td>
</tr>
<tr>
<td>XREF and ATTRIBUTES</td>
<td>Creates a sorted cross-reference listing with attributes.</td>
</tr>
</tbody>
</table>

Using Language Environment runtime options

Several runtime options affect debugging in Language Environment. The TEST runtime option, for example, can be used with a debugging tool to specify the level of control the debugging tool has when the routine being initialized is started. The ABPERC, CHECK, DEPTHCONDLMT, DYNDUMP, ERRCOUNT, HEAPCHK, INTERRUPT, TERMTHDACT, TRACE, TRAP, and USRHDLR options affect condition handling. The ABTERMENC option affects how an application ends (that is, with an abend or with a return code and reason code) when an unhandled condition of severity 2 or greater occurs. Table 9 on page 8 lists the Language Environment runtime options that affect debugging. For a more detailed discussion of these runtime options, see z/OS Language Environment Programming Reference.

Table 9: Language Environment runtime options for debugging.

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABPERC</td>
<td>Specifies that the indicated abend code bypasses the condition handler.</td>
</tr>
<tr>
<td>ABTERMENC</td>
<td>Specifies enclave termination behavior for an enclave ending with an unhandled condition of severity 2 or greater.</td>
</tr>
<tr>
<td>CEEDUMP</td>
<td>Specifies options to control the processing of the Language Environment dump report.</td>
</tr>
<tr>
<td>CHECK</td>
<td>Determines if runtime checking is performed.</td>
</tr>
<tr>
<td>NODEBUG</td>
<td>Controls the COBOL USE FOR DEBUGGING declarative.</td>
</tr>
<tr>
<td>DEPTHCONDLMT</td>
<td>Specifies the limit for the depth of nested synchronous conditions in user-written condition handlers. (Asynchronous signals do not affect DEPTHCONDLMT.)</td>
</tr>
</tbody>
</table>
Table 9: Language Environment runtime options for debugging. (continued)

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYNDUMP</td>
<td>Provides a way to obtain IPCS-readable dumps of user applications that would ordinarily be lost due to the absence of a SYMDUMP, SYSUDUMP, or SYSABEND DD statement.</td>
</tr>
<tr>
<td>ERRCOUNT</td>
<td>Specifies the number of synchronous conditions of severity 2 or greater tolerated. (Asynchronous signals do not affect ERRCOUNT.)</td>
</tr>
<tr>
<td>HEAPCHK</td>
<td>Determines if additional heap check tests are performed.</td>
</tr>
<tr>
<td>HEAPZONES</td>
<td>Activates user heap overlay toleration and checking.</td>
</tr>
<tr>
<td>INFOMSGFILTER</td>
<td>Filters user specified informational messages from the MSGFILE. <strong>Note:</strong> Affects only those messages generated by Language Environment and any routine that calls CEEMSG. Other routines that write to the message file, such as CEEMOUT, do not have a filtering option.</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>Causes Language Environment to recognize attention interrupts.</td>
</tr>
<tr>
<td>MSGFILE</td>
<td>Specifies the ddname of the Language Environment message file.</td>
</tr>
<tr>
<td>MSGQ</td>
<td>Specifies the number of instance specific information (ISI) blocks that are allocated on a per-thread basis for use by an application. Located within the Language Environment condition token is an ISI token. The ISI contains information used by the condition manager to identify and react to a specific occurrence of a condition.</td>
</tr>
<tr>
<td>PROFILE</td>
<td>Controls the use of an optional profiler tool, which collects performance data for the running application. When this option is in effect, the profiler is loaded and the debugger cannot be loaded. If the TEST option is in effect when PROFILE is specified, the profiler tool will not be loaded.</td>
</tr>
<tr>
<td>RPTOPTS</td>
<td>Produces a report that shows the runtime options in effect; see “Determining the runtime options in effect” on page 9.</td>
</tr>
<tr>
<td>RPTSTG</td>
<td>Generates a report of the storage used by an enclave; see “Controlling storage allocation” on page 11.</td>
</tr>
<tr>
<td>STORAGE</td>
<td>Specifies that Language Environment initializes all heap and stack storage to a user-specified value.</td>
</tr>
<tr>
<td>TERMTHDACT</td>
<td>Controls response when an enclave terminates due to an unhandled condition of severity 2 or greater.</td>
</tr>
<tr>
<td>TEST</td>
<td>Specifies the conditions under which a debugging tool assumes control.</td>
</tr>
<tr>
<td>TRACE</td>
<td>Activates Language Environment runtime library tracing and controls the size of the trace table, the type of trace, and whether the trace table should be dumped unconditionally upon termination of the application.</td>
</tr>
<tr>
<td>TRAP</td>
<td>When TRAP is set to ON, Language Environment traps routine interrupts and abends, and optionally prints trace information or invokes a user-written condition handling routine. With TRAP set to OFF, the operating system handles all interrupts and abends. You should generally set TRAP to ON, or your runtime results can be unpredictable.</td>
</tr>
<tr>
<td>USRHDLR</td>
<td>Specifies the behavior of two user-written condition handlers. The first handler specified will be registered at stack frame 0. The second handler specified will be registered before any other user-written condition handlers, once the handler is enabled by a condition.</td>
</tr>
<tr>
<td>XUFLOW</td>
<td>Specifies if an exponent underflow causes a routine interrupt.</td>
</tr>
</tbody>
</table>

**Determining the runtime options in effect**

The runtime options in effect at the time the routine is run can affect routine behavior. Use RPTOPTS(ON) to generate an options report in the Language Environment message file when your routine terminates.
The options report lists runtime options, and indicates where they were set. Figure 1 on page 10 shows a sample options report.

<table>
<thead>
<tr>
<th>LAST WHERE SET</th>
<th>OPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-supplied default</td>
<td>ABPERC(NONE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>ABTERMENC(ABEND)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOAIXBLD</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>ALL3I(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>ANYHEAP(16384,8192,ANYWHERE,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOAUTOTASK</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>BELOWHEAP(8192,4096,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CBOPTS(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CBLSHPOP(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CBQDA(OFF)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CEEDUMP(60,SYSOUT=*,FREE=END,SPIN=UNALLOC)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CHECK(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CHECK(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>COUNTRY(US)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NODEBUG</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>DEPTHCONDLMT(10)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>DYNADUMP(*USERID,NODYNAMIC,TDUMP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>ENVAR(&quot;&quot;)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>FILEHIST</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>FILETAG(NOAUTOCVT,NOAUTOTAG)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INFOMSGFILTER(OFF,,,,)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INQPCOPN</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>LIBSTACK(4096,4096,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGFILE(SYSOUT,FBA,121,0,NOENQ)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGQ(15)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NONONIPTSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>OCSTATUS</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PAGEFRAMESIZE(4K,4K,4K)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NPC</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PLITASKCOUNT(20)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>POSIX(OFF)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PROFILE(OFF,&quot;&quot;)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PRTUNIT(6)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PUNUNIT(7)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>RRUNIT(5)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>RECPAD(OFF)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>RPTOPTS(ON)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>STACK(131072,131072,ANYWHERE,KEEP,524288,131072)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>STORAGE(NONE,NONE,NONE,0)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>TERMTHDACT(TRACE,,96)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>NOTE(ALL,&quot;*&quot;,&quot;PROMPT&quot;,&quot;INSPREF&quot;)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>THREADHEAP(4096,4096,ANYWHERE,KEEP)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>THREADSTACK(OFF,4096,4096,ANYWHERE,KEEP,131072,131072)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>TRACE(OFF,4096,DUMP,LE=0)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>TRAPP(ON,SPIE)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>UPSI(999999999)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>USRSHDLR(,)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>VCTRSAVE(OFF)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>XPLINK(OFF)</td>
</tr>
<tr>
<td>SETCEE command</td>
<td>XFLOW(AUTO)</td>
</tr>
</tbody>
</table>

*Figure 1: Options report example produced by runtime option RPTOPTS(ON)*
Understanding the HEAPZONES and HEAPCHK runtime options

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

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

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

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

HEAPCHK investigates the entire user heap for damage during heap related calls at a frequency based on the specified settings in the option. Because HEAPCHK will traverse the entire user heap, a slow down in application performance will occur. Information about HEAPCHK diagnostic output is discussed in Chapter 3, “Using Language Environment debugging facilities,” on page 33.

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

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

Using the CLER CICS transaction to display and set runtime options

The CICS transaction CLER allows you to display all the current Language Environment runtime options for a region, and to modify a subset of these options. For more information about the CICS CLER transaction, see “Displaying and modifying runtime options with the CLER transaction” on page 319.

Controlling storage allocation

The following runtime options control storage allocation:

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

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

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

Figure 2 on page 12 and Figure 4 on page 14 are examples of storage reports that are produced when RPTSTG(ON) is specified. The sections that follow these reports describe the contents of the reports.

---

**Storage Report for Enclave main 09/17/17 03:31:45 PM**

Language Environment V02 R03.00

**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

---

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

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

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

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

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

---

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

**Figure 4 on page 14** shows an example of a storage report that is produced with XPLINK
Storage Report for Enclave main 09/17/12 03:31:45 PM
Language Environment V02 R03.00

STACK statistics:
Initial size: 131072
Increment size: 131072
Largest used by all concurrent threads: 5416
Number of segments allocated: 1
Number of segments freed: 0

THREADSTACK statistics:
Initial size: 4996
Increment size: 4996
Largest 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: 20480
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
Largest used by any thread: 0
Number of segments allocated: 0
Number of segments freed: 0

THREADHEAP statistics:
Initial size: 4096
Increment size: 4096
Largest used by any thread: 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 4: Storage report produced by RPTSTG(ON) with XPLINK
<table>
<thead>
<tr>
<th>Statistics</th>
<th>Initial size</th>
<th>Increment size</th>
<th>Total heap storage used (sugg. initial size)</th>
<th>Successful Get Heap requests</th>
<th>Successful Free Heap requests</th>
<th>Number of segments allocated</th>
<th>Number of segments freed</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAP24</td>
<td>8192</td>
<td>4096</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ANYHEAP</td>
<td>16384</td>
<td>8192</td>
<td>1139712</td>
<td>487</td>
<td>431</td>
<td>50</td>
<td>36</td>
</tr>
<tr>
<td>BELOWHEAP</td>
<td>8192</td>
<td>4096</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Additional Heap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Heap Pools Statistics:**

- **Pool 1** (size: 8 Get Requests, 3)
  - Successful Get Heap requests: 1-8
  - Successful Get Heap requests: 9-16
  - Successful Get Heap requests: 17-24
  - Successful Get Heap requests: 25-32
  - Successful Get Heap requests: 33-40
  - Successful Get Heap requests: 41-48
  - Successful Get Heap requests: 49-56
  - Successful Get Heap requests: 57-64
  - Successful Get Heap requests: 65-72
  - Successful Get Heap requests: 73-80
  - Successful Get Heap requests: 81-88
  - Successful Get Heap requests: 89-96
  - Successful Get Heap requests: 97-104
  - Successful Get Heap requests: 105-112
  - Successful Get Heap requests: 113-120
  - Successful Get Heap requests: 121-128

- **Pool 2** (size: 32 Get Requests, 268)
  - Successful Get Heap requests: 1-8
  - Successful Get Heap requests: 9-16
  - Successful Get Heap requests: 17-24
  - Successful Get Heap requests: 25-32
  - Successful Get Heap requests: 33-40
  - Successful Get Heap requests: 41-48
  - Successful Get Heap requests: 49-56
  - Successful Get Heap requests: 57-64
  - Successful Get Heap requests: 65-72
  - Successful Get Heap requests: 73-80
  - Successful Get Heap requests: 81-88
  - Successful Get Heap requests: 89-96
  - Successful Get Heap requests: 97-104
  - Successful Get Heap requests: 105-112
  - Successful Get Heap requests: 113-120
  - Successful Get Heap requests: 121-128

- **Pool 3** (size: 128 Get Requests, 186)
  - Successful Get Heap requests: 1-8
  - Successful Get Heap requests: 9-16
  - Successful Get Heap requests: 17-24
  - Successful Get Heap requests: 25-32
  - Successful Get Heap requests: 33-40
  - Successful Get Heap requests: 41-48
  - Successful Get Heap requests: 49-56
  - Successful Get Heap requests: 57-64
  - Successful Get Heap requests: 65-72
  - Successful Get Heap requests: 73-80
  - Successful Get Heap requests: 81-88
  - Successful Get Heap requests: 89-96
  - Successful Get Heap requests: 97-104
  - Successful Get Heap requests: 105-112
  - Successful Get Heap requests: 113-120
  - Successful Get Heap requests: 121-128

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

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

The runtime options should be tuned appropriately to avoid performance problems. See z/OS Language Environment Programming Guide for tips on tuning.

Stack storage statistics

Language Environment stack storage is managed at the thread level; each thread has its own stack-type resources. Table 10 on page 17 describes the fields in the storage report that contain various statistics about stack storage.
Table 10: Storage report fields that display stack storage statistics.

<table>
<thead>
<tr>
<th>Statistics categories</th>
<th>Field contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• STACK</td>
<td>The following fields display statistics for the upward-growing stack.</td>
</tr>
<tr>
<td>• THREADSTACK</td>
<td><strong>Initial size</strong></td>
</tr>
<tr>
<td>• LIBSTACK</td>
<td>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 runtime option.</td>
</tr>
<tr>
<td>• IPT STACK</td>
<td><strong>Increment size</strong></td>
</tr>
<tr>
<td></td>
<td>Size of each incremental segment acquired, as determined by the increment portion of the corresponding runtime option.</td>
</tr>
<tr>
<td>• XPLINK STACK</td>
<td><strong>Maximum used by all concurrent threads</strong></td>
</tr>
<tr>
<td>• XPLINK THREADSTACK</td>
<td>Maximum amount allocated in total at any one time by all concurrently executing threads.</td>
</tr>
</tbody>
</table>

**Largest used by any thread**  
Largest amount allocated ever by any single thread.

**Number of segments allocated**  
Number of segments allocated by all threads which includes the initial segments.

**Number of segments freed**  
Number of incremental segments freed by all threads during the life of the threads. This does not include any incremental segments that were freed during 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 runtime option, but its usage is neither tracked nor reported in the storage report.

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

Heap storage statistics

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

Table 11 on page 18 describes the fields in the storage report that contain various statistics about heap storage. These statistics, in all cases, specify totals for the whole enclave. For THREADHEAP, they indicate the total across all threads in the enclave.

<table>
<thead>
<tr>
<th>Statistics categories</th>
<th>Field contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>THREADHEAP Initial size</td>
<td>Default initial allocation, as specified by the corresponding runtime option. For HEAP24, the initial size is the value of the initSz24 of the HEAP option.</td>
</tr>
<tr>
<td>Increment size</td>
<td>Minimum incremental allocation, as specified by the corresponding runtime option. For HEAP24, the increment size is the value of the incrSz24 of the HEAP option.</td>
</tr>
<tr>
<td>Maximum used by all concurrent threads</td>
<td>Maximum total amount used by all concurrent threads at any one time.</td>
</tr>
<tr>
<td>Largest used by any thread</td>
<td>Largest amount used by any single thread</td>
</tr>
<tr>
<td>Successful Get Heap requests</td>
<td>Number of Get Heap requests.</td>
</tr>
<tr>
<td>Successful Free Heap requests</td>
<td>Number of Free Heap requests.</td>
</tr>
<tr>
<td>Number of segments allocated</td>
<td>Number of incremental segments allocated.</td>
</tr>
<tr>
<td>Number of segments freed</td>
<td>Number of incremental segments individually freed.</td>
</tr>
</tbody>
</table>
### Table 11: Storage report fields that display heap storage statistics. (continued)

<table>
<thead>
<tr>
<th>Statistics categories</th>
<th>Field contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• HEAP</td>
<td>Initial size</td>
</tr>
<tr>
<td>• HEAP24</td>
<td>Default initial allocation, as specified by the corresponding runtime option. For HEAP24, the initial size is the value of the initsz24 of the HEAP option.</td>
</tr>
<tr>
<td>• ANYHEAP</td>
<td>Increment size</td>
</tr>
<tr>
<td>• BELOWHEAP</td>
<td>Minimum incremental allocation, as specified by the corresponding runtime option. For HEAP24, the increment size is the value of the incrsz24 of the HEAP option.</td>
</tr>
<tr>
<td><strong>Total heap storage used</strong></td>
<td>Largest total amount used by the enclave at any one time.</td>
</tr>
<tr>
<td><strong>Successful Get Heap requests</strong></td>
<td>Number of Get Heap requests.</td>
</tr>
<tr>
<td><strong>Successful Free Heap requests</strong></td>
<td>Number of Free Heap requests. The number of Free Heap requests could be less than the number of Get Heap requests if the pieces of heap storage acquired by individual Get Heap requests were not freed individually, but rather were freed implicitly in the course of enclave termination.</td>
</tr>
<tr>
<td><strong>Number of segments allocated</strong></td>
<td>Number of incremental segments allocated.</td>
</tr>
<tr>
<td><strong>Number of segments freed</strong></td>
<td>Number of incremental segments individually freed. The number of incremental segments individually freed could be less than the number allocated if the segments were not freed individually, but rather were freed implicitly in the course of enclave termination.</td>
</tr>
</tbody>
</table>

### 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 callable service.

**Successful Create Heap requests**
Number of successful Create Heap requests.

**Successful Discard Heap requests**
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**
Largest total amount used by the enclave at any one time.

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

**Successful Free Heap requests**
Number of Free Heap requests.

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

**Number of segments freed**
Number of incremental segments individually freed.

### HEAPPOOLS storage statistics

The HEAPPOOLS runtime option (for C/C++ applications only) controls usage of the HEAPPOOLS storage algorithm at the enclave level. The HEAPPOOLS algorithm allows for the definition of one to twelve heap pools, each consisting of a number of storage cells of a specified length. For further details regarding HEAPPOOLS storage statistics in the storage report, see “HEAPPOOLS storage statistics” on page 215.
Modifying condition handling behavior

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

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

Language Environment callable services

Table 12 on page 20 lists the callable services that you can use to modify condition handling. For more information about callable services, see z/OS Language Environment Programming Reference. Note that Fortran programs cannot directly call Language Environment callable services. For more information about how to invoke callable services from Fortran, see Language Environment for MVS & VM Fortran Run-Time Migration Guide.

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

Language Environment runtime options

Table 13 on page 21 shows the Language Environment runtime options that can affect your routine’s condition handling behavior.
## Table 13: Runtime options that modify condition handling.

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ABPERC</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>CHECK</strong></td>
<td>Specifies that checking errors within an application are detected. The Language Environment-conforming languages can define error checking differently.</td>
</tr>
<tr>
<td><strong>DEPTHCONDLMT</strong></td>
<td>Limits the extent to which synchronous conditions can be nested in a user-written condition handler. (Asynchronous signals do not affect DEPTHCONDLMT.) For example, if you specify 5, the initial condition and four nested conditions are processed. If the limit is exceeded, the application terminates with abend code 4091 and reason code 21 (X'15').</td>
</tr>
<tr>
<td><strong>ERRCOUNT</strong></td>
<td>Specifies the number of synchronous conditions of severity 2, 3, and 4 that are tolerated before the enclave terminates abnormally. (Asynchronous signals do not affect ERRCOUNT.) If you specify 0 an unlimited number of conditions is tolerated.</td>
</tr>
<tr>
<td><strong>INTERRUPT</strong></td>
<td>Causes attentions recognized by the host operating system to be passed to and recognized by Language Environment after the environment has been initialized.</td>
</tr>
<tr>
<td><strong>TERMTHDACT</strong></td>
<td>Sets the level of information that is produced when a condition of severity 2 or greater remains unhandled within the enclave. The parameter settings for different levels of information include: • 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.</td>
</tr>
<tr>
<td><strong>TRAP(ON)</strong></td>
<td>Fully Enables the Language Environment condition handler. This causes the Language Environment condition handler to intercept error conditions and routine interrupts. During typical operation, you should use TRAP(ON) when running your applications. When TRAP(ON,NOSPIE) is specified, Language Environment handles all program interrupts and abends through an ESTAE. Use this feature when you do not want Language Environment to issue an ESPIE macro.</td>
</tr>
</tbody>
</table>
Table 13: Runtime options that modify condition handling. (continued)

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAP(OFF)</td>
<td>Disables the Language Environment condition handler from handling abends and program checks/interrupts. ESPIE is not issued with TRAP(OFF), it is still possible to invoke the condition handler through the CEESGL callable service and pass conditions to registered user-written condition handlers. Specify TRAP(OFF) when you do not want Language Environment to issue an ESTAE or an ESPIE. However, TRAP(OFF) can cause several unexpected side effects. For more information, see the TRAP runtime option in z/OS Language Environment Programming Reference. When TRAP(OFF), TRAP(OFF,SPIE), or TRAP(OFF,NOSPIE) is specified and either a program interrupt or abend occurs, the user exit for termination is ignored.</td>
</tr>
<tr>
<td>USRHDLR</td>
<td>Specifies the behavior of two user-written condition handlers. The first handler specified will be registered at stack frame 0. The second handler specified will be registered before any other user-written condition handlers, once the handler is enabled by a condition. When you specify USRHDLR(lastname,supername), lastname gets control at stack frame 0. The supername will get control first, before any user-written condition handlers but after supername has gone through the enablement phase, when a condition occurs.</td>
</tr>
<tr>
<td>XUFLOW</td>
<td>Specifies if an exponent underflow causes a routine interrupt.</td>
</tr>
</tbody>
</table>

**Customizing condition handlers**

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

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

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

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

**Invoking the assembler user exit**

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

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

At enclave initialization, the CEEBXITA user exit runs prior to the enclave establishment. Thus you can modify the environment in which your application runs in the following ways:

- Specify runtime options
- Allocate data sets/files in the user exit
- List abend codes to be passed to the operating system
- Check the values of routine arguments

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

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

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

You can use the assembler user exit to establish enclave termination behavior for an enclave ending with an unhandled condition of severity 2 or greater in the following ways:

- If you do not request an abend in the assembler user exit for the enclave termination call, Language Environment honors the setting of the ABTERMENC option to determine how to end the enclave.
- If you request an abend in the assembler user exit for the enclave termination call, Language Environment issues an abend to end the enclave.

For more information on the assembler user exit, see z/OS Language Environment 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 22 and z/OS Language Environment Programming Guide)
- POSIX signal default action (see z/OS Language Environment Programming Guide)
- The ABTERMENC runtime option (discussed below)

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

If you specify the IBM-supplied default suboption ABEND, Language Environment issues an abend to end the enclave regardless of the setting of the CEEAUE_ABND flag. Additionally, the assembler user exit can alter the abend code, abend reason code, abend dump attribute, and the abend task/step attribute. For more information on using ABTERMENC, see z/OS Language Environment Programming Reference, and for more information on the assembler user exit, see z/OS Language Environment Programming Guide.

If you specify the RETCODE suboption, Language Environment uses the 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 run time, and to display variable values. The process of creating messages and using them requires that you create a message source file, and convert the source file into loadable code for use in your routine.

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

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

Each Language Environment-conforming language also provides ways to display both user-created and runtime messages. (For an explanation of Language Environment runtime messages, see “Interpreting runtime messages” on page 28.)

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

C/C++

For C/C++ routines, output from the printf function is directed to stdout, which is associated with SYSPRINT. All C/C++ runtime messages and perror() messages are directed to stderr. stderr corresponds to the ddname associated with the Language Environment MSGFILE runtime option. The destination of the printf function output can be changed by using the redirection 1>&2 at routine invocation to redirect stdout to the stderr destination. Both streams can be controlled by the MSGFILE runtime option.

COBOL

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

Fortran

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

PL/I

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

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

- To specify the feedback code parameter when calling Language Environment services (see “Using the feedback code parameter” on page 25).
- To code a symbolic feedback code in a user-written condition handler (see “Using the symbolic feedback code” on page 26).

Using the feedback code parameter

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

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

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

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

Language Environment provides callable services that can be used to convert condition tokens to routine variables, messages, or signaled conditions. Table 14 on page 25 lists these callable services and their functions.

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

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

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

For Case 1 condition tokens, Condition_ID is:

<table>
<thead>
<tr>
<th>0 - 15</th>
<th>16 - 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severity Number</td>
<td>Message Number</td>
</tr>
</tbody>
</table>

For Case 2 condition tokens, Condition_ID is:

<table>
<thead>
<tr>
<th>0 - 15</th>
<th>16 - 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Code</td>
<td>Cause Code</td>
</tr>
</tbody>
</table>

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

**Figure 7: Language Environment condition token**

For example, in the condition token: X'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 runtime message to the condition section of the traceback or dump. If a condition is detected when a callable service is invoked without a feedback code, the condition token is passed to the Language Environment condition manager. The condition manager polls active condition handlers for a response. If a condition of severity 0 or 1 remains unhandled, Language Environment resumes without issuing a message. Language Environment does issue messages, however, for COBOL severity 1 conditions. For unhandled conditions of severity 2 or greater, Language Environment issues a message and terminates. For a list of Language Environment runtime messages and corrective information, see [z/OS Language Environment Runtime Messages](https://www.ibm.com). If a second condition is raised while Language Environment is attempting to handle a condition, the message CEE0374C CONDITION = <message_no.> is displayed using a write-to-operator (WTO). The message number in the CEE0374C message indicates the original condition that was being handled when the second condition was raised. This can happen when a critical error is signaled (for example, when internal control blocks are damaged).

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

**Using the symbolic feedback code**

The symbolic feedback code represents the first 8 bytes of a 12-byte condition token. You can think of the symbolic feedback code as the nickname for a condition. As such, the symbolic feedback code can be used in user-written condition handlers to screen for a given condition, even if it occurs at different locations in an application. For more details on symbolic feedback codes, see [z/OS Language Environment Programming Guide](https://www.ibm.com).
Chapter 2. Classifying errors

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

Identifying problems in routines

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

Language Environment module names

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

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

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

Common errors in routines

These common errors have simple solutions:

- If you do not have enough virtual storage, increase your region size or decrease your storage usage (stack size) by using the storage-related runtime options and callable services. (See “Controlling storage allocation” on page 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 these items, 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 that are shared between Fortran routines and C library routines can produce unexpected results. Language Environment provides several cataloged procedures to properly resolve duplicate names. For more information on how to avoid name conflicts, see z/OS Language Environment Programming Guide.

Changes in optimization levels, addressing modes, and input/output file formats can also cause unanticipated problems in your routine.

In most cases, generated condition tokens or runtime messages point to the nature of the error. The runtime messages offer the most efficient corrective action. To help you analyze errors and determine the most useful method to fix the problem, Table 15 on page 28 lists common error symptoms, possible causes, and programmer responses.

© Copyright IBM Corp. 1991, 2018
### Table 15: Common error symptoms, possible causes, and programmer responses

<table>
<thead>
<tr>
<th>Error Symptom</th>
<th>Possible cause</th>
<th>Programmer response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbered runtime message appears</td>
<td>Condition that is raised in routine.</td>
<td>For any messages you receive, read the Programmer Response. For information about message structure, see “Interpreting runtime messages” on page 28.</td>
</tr>
<tr>
<td>User abend code &lt; 4000</td>
<td>• A non-Language Environment abend occurred.</td>
<td>See the Language Environment abend codes in z/OS Language Environment Runtime Messages. Check for a subsystem-generated abend or a user-specified abend.</td>
</tr>
<tr>
<td></td>
<td>• The assembler user exit requested an abend for an unhandled condition of severity ≥2.</td>
<td></td>
</tr>
<tr>
<td>User abend code ≥ 4000</td>
<td>• Language Environment detected an error and could not proceed.</td>
<td>For any abends you receive, read the appropriate explanation that is listed in the abend codes section of z/OS Language Environment Runtime Messages.</td>
</tr>
<tr>
<td></td>
<td>• An unhandled software-raised condition occurred and ABTERMENC(ABEND) was in effect.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The assembler user exit requested an abend for an unhandled condition of severity 4.</td>
<td></td>
</tr>
<tr>
<td>System abend with TRAP(OFF)</td>
<td>Cause depends on type of malfunction.</td>
<td>Respond appropriately. See the messages and codes book of the operating system.</td>
</tr>
<tr>
<td>System abend with TRAP(ON)</td>
<td>System-detected error.</td>
<td>See the messages and codes information of the operating system.</td>
</tr>
<tr>
<td>No response (wait/loop)</td>
<td>Application logic failure.</td>
<td>Check routine logic. Ensure that ERRCOUNT and DEPTHCONDLMT runtime options are set to a nonzero value.</td>
</tr>
<tr>
<td>Unexpected message (message received was not from most recent service)</td>
<td>Condition that is caused by something that is related to current service.</td>
<td>Generate a traceback by using CEE3DMP.</td>
</tr>
<tr>
<td>Incorrect output</td>
<td>Incorrect file definitions, storage overlay, incorrect routine mask setting, references to uninitialized variables, data input errors, or application routine logic error.</td>
<td>Correct the appropriate parameters.</td>
</tr>
<tr>
<td>No output</td>
<td>Incorrect ddname, file definitions, or message file setting.</td>
<td>Correct the appropriate parameters.</td>
</tr>
<tr>
<td>Nonzero return code from enclave</td>
<td>Unhandled condition of severity 2, 3, or 4, or the return code was issued by the application routine.</td>
<td>Check the Language Environment message file for runtime message.</td>
</tr>
<tr>
<td>Unexpected output</td>
<td>Conflicting library module names.</td>
<td>See the name conflict resolution steps that are outlined in z/OS Language Environment Programming Guide.</td>
</tr>
</tbody>
</table>

### Interpreting runtime messages

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

- On z/OS, runtime messages are written by default to ddname SYSOUT. If SYSOUT is not specified, then the messages are written to SYSOUT=*.
- On CICS, the runtime messages are written to the CESE transient data QUEUE.
The default message file ddname can be changed by using the MSGFILE runtime option. For information about displaying runtime messages for Language Environment, COBOL, Fortran, or PL/I routines, see z/OS Language Environment Programming Guide.

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

In the following example Language Environment message:

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

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

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

### Message prefix

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

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

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

### Message number

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

### Severity code

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

Understanding abend codes

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

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

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

You can specify the ABTERMENC runtime option to determine what action is taken when an unhandled condition of severity 2 or greater occurs. For more information on ABTERMENC, see “Establishing enclave termination behavior for unhandled conditions” on page 23, as well as z/OS Language Environment Programming Reference.

User abends

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

System abends

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

When a system abend occurs, the operating system can generate a system dump. System dumps are written to ddname SYSMDUMP, SYSABEND, or SYSUDUMP. If the DYNDUMP runtime option is used in combination with the TERMTHDACT runtime option, the system dump can be written without the ddname specified. System dumps show the memory state at the time of the condition. See “Generating a system dump” on page 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 runtime library or edcmtext can be invoked directly with the errno2 value as input.

Format

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

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

The errno2_value is specified as 8 hexadecimal characters.

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

Usage notes

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

Message returns

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

Usage: edcmtext errno2_value

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

errno2_value: No information is currently available for this errno2_value.

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

Usage: edcmtext errno2_value

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

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

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

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

Examples

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

JrEdc1opsEinval01: The mode argument passed to fopen() or freopen() did not begin with r, w, or a.

Action: Correct the mode argument. The first keyword of the mode argument must be the open mode. Ensure the open mode is specified first and begins with r, w, or a.

Source: edc1opst.c
Exit Values

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

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

Debug tools

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

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

You can use dbx to debug Language Environment applications, including C/C++ programs. z/OS UNIX System Services Command Reference has information on dbx subcommands, while z/OS UNIX System Services Programming Tools contains usage information.

Language Environment dump service, CEE3DMP

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

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

Generating a Language Environment dump with CEE3DMP

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

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

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

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

The syntax for CEE3DMP is:

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

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

`options`
A 255-byte fixed-length character string that contains options that describe the type, format, and destination of dump information. The options are declared as a string of keywords that are separated by blanks or commas. Some options also have suboptions that follow the option keyword, and are contained in parentheses. The last option declaration is honored if there is a conflict between it and any preceding options. Table 16 on page 34 lists the CEE3DMP options and related information.

The IBM-supplied default settings for CEE3DMP are:

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

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

Table 16 on page 34 summarizes the dump options available to CEE3DMP. For more information about the CEE3DMP callable service and dump options, see z/OS Language Environment Programming Reference. For an example of a Language Environment dump, see “Understanding the Language Environment dump” on page 40.

**Table 16: CEE3DMP options.**

<table>
<thead>
<tr>
<th>Dump options</th>
<th>Abbreviation</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLAVE(ALL)</td>
<td>ENCL</td>
<td>Dumps all enclaves associated with the current process. (In ILC applications in which a C/C++ routine calls another member language routine, and that routine in turn calls CEE3DMP, traceback information for the C/C++ routine is not provided in the dump.) This is the default setting for ENCLAVE.</td>
</tr>
<tr>
<td>ENCLAVE(CURRENT)</td>
<td>ENCL(CUR)</td>
<td>Dumps the current enclave.</td>
</tr>
</tbody>
</table>

On CICS, only ENCLAVE(CURRENT) and ENCLAVE(1) settings are supported.
<table>
<thead>
<tr>
<th>Dump options</th>
<th>Abbreviation</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLAVE(n)</td>
<td>ENCL(n)</td>
<td>Dumps a fixed number of enclaves, indicated by n.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>On CICS, only ENCLAVE(CURRENT) and ENCLAVE(1) settings are supported.</strong></td>
</tr>
<tr>
<td>THREAD(ALL)</td>
<td>THR(ALL)</td>
<td>Dumps all threads in this enclave (including in a PL/I multitasking environment).</td>
</tr>
<tr>
<td>THREAD(CURRENT)</td>
<td>THR(CUR)</td>
<td>Dumps the current thread in this enclave.</td>
</tr>
<tr>
<td>TRACEBACK</td>
<td>TRACE</td>
<td>Includes a traceback of all active routines. The traceback shows transfer of control from calls or exceptions. Calls include PL/I transfers of control from BEGIN-END blocks or ON-units.</td>
</tr>
<tr>
<td>NOTRACEBACK</td>
<td>NOTRACE</td>
<td>Does not include a traceback of all active routines.</td>
</tr>
<tr>
<td>FILES</td>
<td>FILE</td>
<td>Includes attributes of all open files. File control blocks are included when the BLOCKS option is also specified. File buffers are included when the STORAGE option is specified.</td>
</tr>
<tr>
<td>NOFILES</td>
<td>NOFILE</td>
<td>Does not include file attributes.</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>VAR</td>
<td>Includes a symbolic dump of all variables, arguments, and registers.</td>
</tr>
<tr>
<td>NOVARIABLES</td>
<td>NOVAR</td>
<td>Does not include variables, arguments, and registers.</td>
</tr>
<tr>
<td>BLOCKS</td>
<td>BLOCK</td>
<td>Dumps control blocks from Language Environment and member language libraries. Global control blocks, as well as control blocks associated with routines on the call chain, are printed. Control blocks are printed for the routine that called CEE3DMP. The dump proceeds up the call chain for the number of routines that are specified by the STACKFRAME option. Control blocks for files are also dumped if the FILES option was specified. See the FILES option for more information. If the TRACE runtime option is set to ON, the trace table is dumped if BLOCKS is specified. If the Heap Storage Diagnostics report is requested using the HEAPCHK runtime option, the report is displayed when BLOCKS is specified.</td>
</tr>
<tr>
<td>NOBLOCKS</td>
<td>NOBLOCK</td>
<td>Does not include control blocks.</td>
</tr>
<tr>
<td>STORAGE</td>
<td>STOR</td>
<td>Dumps the storage used by the routine. The number of routines dumped is controlled by the STACKFRAME option.</td>
</tr>
<tr>
<td>NOSTORAGE</td>
<td>NOSTOR</td>
<td>Suppresses storage dumps.</td>
</tr>
<tr>
<td>STACKFRAME(ALL)</td>
<td>SF(ALL)</td>
<td>Dumps all stack frames from the call chain. This is the default setting for STACKFRAME.</td>
</tr>
<tr>
<td>STACKFRAME(n)</td>
<td>SF(n)</td>
<td>Dumps a fixed number of stack frames, indicated by n, from the call chain. The specific information dumped for each stack frame depends on the VARIABLES, BLOCK, and STORAGE options declarations. The first stack frame dumped is the caller of CEE3DMP, followed by its caller, and proceeding backward up the call chain.</td>
</tr>
<tr>
<td>PAGESIZE(n)</td>
<td>PAGE(n)</td>
<td>Specifies the number of lines, n, on each page of the dump.</td>
</tr>
<tr>
<td>FNAME(s)</td>
<td>FNAME(s)</td>
<td>Specifies the ddname of the file to which the dump is written.</td>
</tr>
</tbody>
</table>
### Table 16: CEE3DMP options. (continued)

<table>
<thead>
<tr>
<th>Dump options</th>
<th>Abbreviation</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONDITION</td>
<td>COND</td>
<td>Dumps condition information for each condition active on the call chain.</td>
</tr>
<tr>
<td>NOCONDITION</td>
<td>NOCOND</td>
<td>For each condition active on the call chain, does not dump condition information.</td>
</tr>
<tr>
<td>ENTRY</td>
<td>ENT</td>
<td>Includes a description of the program unit that called CEE3DMP and the registers on entry to CEE3DMP.</td>
</tr>
<tr>
<td>NOENTRY</td>
<td>NOENT</td>
<td>Does not include a description of the program unit that called CEE3DMP or registers on entry to CEE3DMP.</td>
</tr>
<tr>
<td>GENOPTS</td>
<td>GENO</td>
<td>Generates a runtime options report in the dump output. This will be the default if an unhandled condition occurs, and a CEEDUMP is generated due to the setting of the TERMTHDACT runtime option setting.</td>
</tr>
<tr>
<td>NOGENOPTS</td>
<td>NOGENO</td>
<td>Does not generate a runtime options report in the dump output. NOGENOPTS is the default for user-called dumps.</td>
</tr>
<tr>
<td>REGSTOR(reg_stor_amount)</td>
<td>REGST(reg_stor_amount)</td>
<td>Controls the amount of storage to be dumped around registers. Default is 96 bytes. Specify REGSTOR(0) if no storage around registers is required.</td>
</tr>
</tbody>
</table>

### Generating a Language Environment dump with TERMTHDACT

The TERMTHDACT runtime option produces a dump during program checks, abnormal terminations, or calls to the CEESGL service. You must use TERMTHDACT(DUMP) in conjunction with TRAP(ON) to generate a Language Environment dump. You can use TERMTHDACT to produce a traceback, Language Environment dump, or user address space dump when a thread ends abnormally because of an unhandled condition of severity 2 or greater. If this is the last thread in the process, the enclave goes away. A thread terminating in a non-POSIX environment is analogous to an enclave terminating because Language Environment Version 1 supports only single threads. For information on enclave termination, see z/OS Language Environment Programming Guide.

The TERMTHDACT suboptions QUIET, MSG, TRACE, DUMP, UAONLY, UATRACE, UADUMP, and UAIMM control the level of information available. Table 17 on page 36 lists the suboptions, the levels of information produced, and the destination of each.

### Table 17: TERMTHDACT suboptions, level of information, and destinations

<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of Information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIET</td>
<td>No information</td>
<td>No destination.</td>
</tr>
<tr>
<td>MSG</td>
<td>Message</td>
<td>Terminal or ddname specified in MSGFILE runtime option.</td>
</tr>
<tr>
<td>TRACE</td>
<td>Message and Language Environment dump containing only a traceback</td>
<td>Message goes to terminal or ddname specified in MSGFILE runtime option. Traceback goes to CEEDUMP file.</td>
</tr>
<tr>
<td>DUMP</td>
<td>Message and complete Language Environment dump</td>
<td>Message goes to terminal or ddname specified in MSGFILE runtime option. Language Environment dump goes to CEEDUMP file.</td>
</tr>
<tr>
<td>Suboption</td>
<td>Level of Information</td>
<td>Destination</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>UAONLY</td>
<td>SYSMDUMP, SYSABEND dump, or SYSUDUMP depending on the DD card used in the JCL in z/OS. 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 runtime option. Traceback goes to CEEDUMP file. Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified; for CICS the transaction dump goes to DFHDMPA or the DFHDMPB data set.</td>
</tr>
<tr>
<td>UADUMP</td>
<td>Message, Language Environment dump, and SYSMDUMP, SYSABEND dump, or SYSUDUMP depending on the DD card used in the JCL in z/OS. In CICS, a transaction dump is created.</td>
<td>Message goes to terminal or ddname specified in MSGFILE runtime option. Language Environment dump goes to CEEDUMP file. Language Environment generates a U4039 abend which allows a system dump of the user address space to be generated. For z/OS, the system dump is written to the ddname specified; for CICS the transaction dump goes to DFHDMPA or the DFHDMPB data set.</td>
</tr>
<tr>
<td>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,PIE) is in effect, UAIMM yields UAONLY behavior. For software raised conditions or signals, UAIMM behaves the same as UAONLY.</td>
<td>Message goes to terminal or ddname specified in MSGFILE runtime option. User address space dump goes to ddname specified for z/OS; or a CICS transaction dump goes to the DFHDMPA or DFHDMPB data set.</td>
</tr>
</tbody>
</table>

The TRACE and UATRACE suboptions of TERMTHDACT use these dump options:
- CONDITION
- ENCLAVE(ALL)
- FILES
- FNAME(CEEDUMP)
- GENOPTS
- NOBLOCKS
The DUMP and UADUMP suboptions of TERMTHDACT use these dump options:

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

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

Considerations for setting TERMTHDACT options

The output of TERMTHDACT may vary depending upon which languages and subsystems are processing the request. This section describes the considerations associated with issuing the TERMTHDACT suboptions. For more information about the TERMTHDACT runtime option, see z/OS Language Environment Programming Reference.

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

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

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

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

- **CICS Considerations**
  - TERMTHDACT output is written to a transient data queue named CESE, or to the CICS transaction dump, depending on the setting of the CESE|CICSSDSS suboption of the TERMTHDACT runtime option. Table 18 on page 39 shows the behavior of CESE|CICSSDSS when they are used with the other suboptions of TERMTHDACT.
  - Because Language Environment does not own the ESTAE, the suboption UAIMM will be treated as UAONLY.
  - All associated Language Environment dumps will be suppressed if termination processing is the result of an EXEC CICS ABEND with NODUMP.
  - Program checks and other abends will cause CICS to produce a CICS transaction dump.

---

<table>
<thead>
<tr>
<th>Options</th>
<th>TERMTHDACT(X,CESE,)</th>
<th>TERMTHDACT(X,CICSSDSS,)</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    | CEECDUMP to CESE queue followed by U4038 abend with nodump option. | CEEDUMP to new transaction server queue which CICS will read and write to CICS transaction dump later.  
  • U4039 abend to force CICS transaction dump followed by U4038 abend with nodump option.  
  • Message to CESE or MSGFILE. |
| UATRACE | U4039 abend with traceback to CESE queue followed by U4038 abend with nodump option. | • Language Environment will write traceback, variables, COBOL working storage, C writeable statics. The member handlers will be invoked to provide the desired output to the new transaction server queue (which CICS will read and write to CICS transaction dump later).  
  • U4039 abend to force CICS transaction dump followed by U4038 abend with nodump option.  
  • Message to CESE or MSGFILE. |
| UADUMP  | U4039 abend with CEECDUMP to CESE queue followed by U4038 abend with nodump option. | CEECDUMP 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. |
Table 18: Condition handling of 0Cx abends (continued)

<table>
<thead>
<tr>
<th>Options</th>
<th>TERMTHDACT(X,CESE, )</th>
<th>TERMTHDACT(X,CICSDDS, )</th>
</tr>
</thead>
</table>
| UAONLY  | U4039 abend followed by U4038 abend with nodump option. | • U4039 abend followed by U4038 abend with nodump option.  
  • No CEEDUMP information is generated.  
  • Same as CESE. |
| UAIMM  | U4039 abend followed by U4038 abend with nodump option. | • U4039 abend followed by U4038 abend with nodump option.  
  • No CEEDUMP information is generated.  
  • Same as CESE. |

Generating a Language Environment dump with language-specific functions

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

C/C++ routines can use the functions cdump(), csnap(), and ctrace() to produce a Language Environment dump. All three functions call the CEE3DMP callable service, and each function includes an options string consisting of different CEE3DMP options that you can use to control the information contained in the dump. For more information on these functions, see “Generating a Language Environment dump of a C/C++ routine” on page 178.

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 247.

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 268.

Understanding the Language Environment dump

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

This sample illustrates a dump for enclave main. The example assumes full use of the CEE3DMP dump options. Ellipses are used to summarize some sections of the dump and information regarding unhandled conditions may not be present at all. Sections of the dump are numbered to correspond with the descriptions given in “Sections of the Language Environment dump” on page 54.

The CEE3DMP was generated by the C program CELSAMP shown in Figure 8 on page 41. CELSAMP uses the DLL CELDLL shown in Figure 11 on page 44.
Figure 8: The C program CELSAMP
main() {
    dllhandle * handle;
    int     i = 0;
    FILE*   fp1;
    FILE*   fp2;
    _FEEDBACK fc;
    _INT4    token;
    _ENTRY   pgmptr;

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

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

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

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

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

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

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

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

Figure 9: The C program CELSAMP (continued)
printf("Register condition handler...\n");
pgmprtr.nesting = NULL;
token = 2;
CEEHDLR (&pgmprtr, &token, &fc);
if (!_FBCHECK(fc, CEE000)) {
    printf("CEEHDLR failed with message number %d\n", fc.tok_msgno);
    exit(108);
}
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("Divide by zero...\n");
i = 1/i;
printf("Error -- Should not get here\n");
exit(110);
}

Figure 10: The C program CELSAMP (continued)
/* DLL containing Condition Handler that takes dump and percolates */
#pragma options(SERVICE("1.3.b.0001"),TEST(SYM),NOOPT)
#pragma export(dump_n_perc)
#include <stdio.h>
#include <leawi.h>
#include <stdlib.h>
#include <string.h>
#include <ceeedcct.h>
char wsa_array[10] = { 'C','E','L','D','L','L',' ','W','S','A'};
#define OPT_STR "THREAD(ALL) BLOCKS STORAGE GENOPTS"
#define TITLE_STR "Sample dump produced by calling CEE3DMP"

void dump_n_perc(_FEEDBACK *cond,_INT4 *input_token,
        _INT4 *result, _FEEDBACK *new_cond) {
    /* values for handling the conditions */
    #define percolate   20
    _CHAR80 title;
    _CHAR255 options;
    _FEEDBACK fc;
    printf(">>> dump_n_perc: Msg # is %d\n",cond->tok_msgno);
    /* check if the DIVIDE-BY-ZERO message (0C9) */
    if (cond->tok_msgno == 3209) {
        memset(options,' ',sizeof(options));
        memcpy(options,OPT_STR,sizeof(OPT_STR)-1);
        memset(title,' ',sizeof(title));
        memcpy(title,TITLE_STR,sizeof(TITLE_STR)-1);
        printf(">>> dump_n_perc: Taking dump\n");
        CEE3DMP(title,options,&fc);
        if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
            printf("CEE3DMP failed with msgno %d\n",fc.tok_msgno);
            exit(299);
        }
    }
    *result = percolate;
    printf(">>> dump_n_perc: Percolating condition\n");
}

Figure 11: The C DLL CELDLL

For easy reference, the sections of the following dump are numbered to correspond with the descriptions in "Sections of the Language Environment dump" on page 54.
Using Language Environment debugging facilities 45
Condition Information for Active Routines

Condition Information for //POSIX.CRTL.C(CELSAMP)' (DSA address 265E9208)

CIB Address: 265E9208

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

Location:
Program Unit: //POSIX.CRTL.C(CELSAMP)' Entry: main Statement: 150 Offset: +000009BA

Machine State:
ILC...... 0002 Interruption Code...... 0009
PSW..... 078D2400 A5E00A7C
GPR0..... 00000000_00000000  GPR1..... 00000000_A647FE0A GPR2..... 00000000_265E92C5 GPR3..... 00000000_265E92C2
GPR4..... 00000000_265E92C2 GPR5..... 00000000_265E92C2 GPR6..... 00000000_265E92C2 GPR7..... 00000000_25E000FA
GPR8..... 00000000_265E92C2 GPR9..... 00000000_25E000FA GPR10.... 00000000_265E92C2 GPR11.... 00000000_265E92C2
GPR12.... 00000000_265E92C2 GPR13.... 00000000_265E92C2 GPR14.... 00000000_265E92C2 GPR15.... 00000000_265E92C2

Storage dump near condition, beginning at location: 25E00A6A
+0000 25E00A6A  4400C1AC 5800D0AC 41600001 8E600020  1D601807 5000D0AC 4400C1AC 58F039E2
|..A......-...-..&.....A..0.S|

GPREG STORAGE:
Storage around GPR0 (00000000)
+0020 00000020    Inaccessible storage.
+0040 00000040    Inaccessible storage.

Local Variables:
title[0..6]      unsigned char         'S'     'a'     'm'     'p'     'l'     'e'     ' '
title[7..13]                           'd'     'u'     'm'     'p'     ' '     'p'     'r'

options[0..6]    unsigned char         'T'     'H'     'R'     'E'     'A'     'D'     '('

options[28..34]                        'O'     'C'     'K'     'S'     ' '     'S'     'T'

options[35..41]                        'E'     'N'     'O'     'P'     'T'     'S'     ' '

options[42..48] to options[245..251] elements same as above.

options[252..254]                       ' '     ' '     ' '     ' '     ' '     ' '     ' '

fc               struct _FEEDBACK

__func__[0..6]   static unsigned char

z/OS: Language Environment Debugging Guide
Using Language Environment debugging facilities
[6] Information for thread 2625970000000001

Registers on Entry to CEE3DMP:
PM....... 0100
GPR0..... 00000000_00000001  GPR1..... 00000000_26935D78  GPR2..... 00000000_2671C43C  GPR3..... 00000000_26935D78
GPR4..... 00000000_25E00E00  GPR5..... 00000000_26935D78  GPR6..... 00000000_2660A930  GPR7..... 00000000_268E6930
GPR8..... 00000000_26935D78  GPR9..... 00000000_268E6930  GPR10.... 00000000_26935D78  GPR11.... 00000000_26935D78
GPR12.... 00000000_26935D78  GPR13.... 00000000_26935D78  GPR14.... 00000000_26935D78  GPR15.... 00000000_26935D78
FPR0..... 4D000000  00000BD1            FPR2..... 00000000  00000000  FPR4..... 00000000  00000000  FPR6..... 00000000  00000000
GPREG STORAGE:
Storage around GPR0 (00000001)
-0001 00000000    Inaccessible storage.
+001F 00000020    Inaccessible storage.
+003F 00000040    Inaccessible storage.

[7] Traceback:

DSA   Entry       E  Offset  Statement   Load Mod            Program Unit                   Service  Status
1     CEEOPML2    +00000F90              CEEPLPKA            CEEOPML2                       HLE7770  Call
2     EDCOWRP2    +00000F38              CEEEV003                                                    Call
3     thread_func +000000AE  47          CELSAMP             CELSAMP                        1.1.D    Call
4     CEEOPCMM    +00000986              CEEBINIT            CEEOPCMM                       HLE7770  Call

DSA   DSA Addr   E  Addr    PU Addr    PU Offset  Comp Date  Compile Attributes
1     26938090   25F5E148   25F5E148   +00000F90  20100319   CEL       POSIX
2     26937DE0   26410CA4   2640FA00   +000021DC  20100319   LIBRARY   POSIX
3     26937D38   25E00D88   25E00D88   +000000AE  20070105   C/C++     POSIX EBCDIC  HFP
4     2697DFF0   0000C5D8   0000C5D8   +00000986  20100319   CEL       POSIX

Fully Qualified Names
DSA   Entry       Program Unit                               Load Module
3     thread_func //'POSIX.CRTL.C(CELSAMP)'                  CELSAMP

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

CEEOPML2 (DSA address 26938090):
UPSTACK DSA
Saved Registers:
GPR0..... 00000001  GPR1..... 26935D78  GPR2..... 2671C43C  GPR3..... 26935D78
GPR4..... D96CA288  GPR5..... 00000000  GPR6..... 265E61A0  GPR7..... 2660A930
thread_func (DSA address 26937D38):
UPSTACK DSA
Parameters:
parm             void *             0x25E00ED0
Saved Registers:
GPR0..... 00000001  GPR1..... 26935D78  GPR2..... 2671C43C  GPR3..... 26935D78
GPR4..... D96CA288  GPR5..... 00000000  GPR6..... 265E61A0  GPR7..... 2660A930

[10] Control Blocks for Active Routines:

DSA for CEEOPML2: 26938090
+000000  FLAGS.... 0000      member... CCCC      BKC...... 26937DE0  FWC...... 26938210  R14...... A5F5EF76
+000010  R15...... A5F60090  R0....... 25F5F2D4  R1....... 26938114  R2....... 2671C43C  R3....... 26935D78
+000024  R4....... 00000000  R5....... 2660A954  R6....... 265E61A0  R7....... 2660A930  R8....... 25E00ED0

[11] Storage for Active Routines:

DSA frame: 26937DE0
+000000 26937DE0  10CCCCCC 26937D38 26938090 A6411BDE  A5F5E148 25E00ED0 26938210 26938210 R14...... A5F5EF76
+000010 R15...... A5F60090 R0....... 25F5F2D4 R1....... 26938114 R2....... 2671C43C R3....... 26935D78
+000024 R4....... 00000000 R5....... 2660A954 R6....... 265E61A0 R7....... 2660A930 R8....... 25E00ED0

[12] Control Blocks Associated with the Thread:

CAA: 26936BD8
+000000 26936BD8  00000800 00000000 26937D20 00000000  00000000 00000000 00000000 00000000

[13] Enclave variables:

\*\*\*\C(CELSAMP)\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\
...
HeapCHK Element Table (HCEL) for Heapid 26999FCC:

- Header: 2699D028

<table>
<thead>
<tr>
<th>Address</th>
<th>Seg Addr</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

HeapCHK Element Table (HCEL) for Heapid 00000000:

- Header: 265E6228

<table>
<thead>
<tr>
<th>Address</th>
<th>Seg Addr</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heap Storage Diagnostics:

<table>
<thead>
<tr>
<th>Stg Addr</th>
<th>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>269DF020</td>
<td>25FC9358</td>
<td>+00000000</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>269DF032</td>
<td>+000013</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>269DF0B0</td>
<td>+00000072</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>263A2BC8</td>
<td>+00000000</td>
<td>CEEEV003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>263903B0</td>
<td>+000005E4</td>
<td>CEEEV003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>263BCCC0</td>
<td>+00000248</td>
<td>CEEEV003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>261A6430</td>
<td>+00000224</td>
<td>CEEEV003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>261A6740</td>
<td>+0000006C</td>
<td>CEEEV003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2619074A</td>
<td>+0000121A</td>
<td>CEEEV003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25E000C0</td>
<td>+000007BC</td>
<td>CELSAMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2659CF6E</td>
<td>-FF87A878</td>
<td>CEEEV003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25E92F60</td>
<td>+000001B8</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25E000C0</td>
<td>+000007BC</td>
<td>CELSAMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2659CF6E</td>
<td>-FF87A878</td>
<td>CEEEV003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25E92F60</td>
<td>+000001B8</td>
<td>CEEPLPKA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Language Environment Trace Table:

- Most recent trace entry is at displacement: 002980

<table>
<thead>
<tr>
<th>Displacement Trace Entry in Hexadecimal</th>
<th>Trace Entry in EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>Time 15.25.14.872201 Date 2010.04.07 Thread ID... 2625860000000000</td>
</tr>
<tr>
<td>+000010</td>
<td>Member ID... 03 Flags... 000000 Entry Type... 0000001</td>
</tr>
<tr>
<td>+000038</td>
<td>60606E4D F0F8F55D 40979989 95A3864D 5D404040 40404040 40404040 40404040 40404040</td>
</tr>
<tr>
<td>+000058</td>
<td>40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040</td>
</tr>
<tr>
<td>+000078</td>
<td>40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040 40404040</td>
</tr>
<tr>
<td>+000080</td>
<td>Time 15.25.14.998940 Date 2010.04.07 Thread ID... 2625860000000000</td>
</tr>
<tr>
<td>+000098</td>
<td>4C60664D F0F8F55D 40D9F1F5 7E0F0F0F 00000000</td>
</tr>
<tr>
<td>+0000BB</td>
<td>F0F8F55D 40D9F1F5 7E0F0F0F 00000000</td>
</tr>
</tbody>
</table>

Using Language Environment debugging facilities 51
[15]Enclave Storage:
Initial (User) Heap                                         : 26609018
HANC..*...*.....w-...-S-........|
+000000 26609018  C8C1D5C3 25E15C90 25E15C90 00000000  A6609018 2660E260 00000000 000020BB |
|................................|
+0045E0 2660D5F8 - +00523F 26611017             same as above
+005240 26612000  C8C1D5C3 2671E000 265E5000 00000000 00100028 00000000 000007B0 |
|................................|
+003860 265E8660  BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB |
|................................|
|................................|
+003800 265E8860 - +003FFF 265E8FFFF same as above
LE/370 Anywhere Heap                                        : 26612000
HANC....;&...*../..............|
+000000 26612000  C8C1D5C3 2671E000 265E5000 25E15C00 25E15C00 00000000 00100028 00000000 |
|................................|
+002900 26612020  00000008 C5CB773F 719897EB 26258600 00000000 03000000 00000001 |
|................................|
+000000 26612040  94818995 40404040 40404040 40404040 40404040 40404040 40404040 |
|................................|
+000000 26612040  00040000 00000003 00000000 00000000 00000000 00000000 00000000 |
|................................|
+000000 26612040  00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
|................................|
+000000 26612040  00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
|................................|
+000000 26612040  00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
|................................|
+000000 26612040  00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
|................................|
+000000 26612040  00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
|................................|
+000000 26612040  00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
|................................|
+000000 26612040  00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
|................................|
+000000 26612040  00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
<p>|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|
|................................|</p>
<table>
<thead>
<tr>
<th>LAST WHERE SET</th>
<th>OPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-supplied default</td>
<td>ABPERC(NONE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>ABTERMENC(ABEND)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOAIXBLD</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>ALL31(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>ANYHEAP(16384,8192,ANYWHERE,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOAUTOTASK</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>BELOWHEAP(8192,4096,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CBLOPTS(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CBLPSPHOP(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CBLOQA(OFF)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>CEEDEMP(0, SYSOUT=*, FREE=END, SPIN=UNALLOC)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>CHECK(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>COUNTRY(US)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NODEBUG</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>DEPTHCONDLMT(10)</td>
</tr>
<tr>
<td>DD:CEEOPTS</td>
<td>DYNDEMP(POSIX.DEBUGG.HLE7770, DYNAMIC, )</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>ERRCOUNT(0)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>ERRUNIT(6)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>FILEHIST</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>FILETAG(NOAUTOCVT, NOAUTOTAG)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>FILETAG(NOAUTOCVT, NOAUTOTAG)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>HEAP(32768,32768, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NONONIPTSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PAGEFRAMESIZE(4K, 4K, 4K)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOPC</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PLOPCP(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INFOMSGFILTER(OFF,,,)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>IIQFCOPN</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>LIBSTACK(4096,4096,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGFILE(SYSTOUT,FBA,121,0,NOENQ)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGQ(15)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOTEREUS</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOSIMVRD</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STAC(131072, 131072, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STORAGE(AA, BB, CC, 0)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>TERMTHDACT(UADUMP,,96)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NORTEREUS</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOSIMVRD</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STACK(131072, 131072, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STORAGE(AA, BB, CC, 0)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>TERMTHDACT(UADUMP,,96)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>HEAP(32768,32768, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NONONIPTSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PAGEFRAMESIZE(4K, 4K, 4K)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOPC</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PLOPCP(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INFOMSGFILTER(OFF,,,)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>IIQFCOPN</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>LIBSTACK(4096,4096,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGFILE(SYSTOUT,FBA,121,0,NOENQ)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGQ(15)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOTEREUS</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOSIMVRD</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STAC(131072, 131072, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STORAGE(AA, BB, CC, 0)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>TERMTHDACT(UADUMP,,96)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NORTEREUS</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOSIMVRD</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STACK(131072, 131072, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STORAGE(AA, BB, CC, 0)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>TERMTHDACT(UADUMP,,96)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>HEAP(32768,32768, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NONONIPTSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PAGEFRAMESIZE(4K, 4K, 4K)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOPC</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PLOPCP(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INFOMSGFILTER(OFF,,,)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>IIQFCOPN</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>LIBSTACK(4096,4096,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGFILE(SYSTOUT,FBA,121,0,NOENQ)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGQ(15)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOTEREUS</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOSIMVRD</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STAC(131072, 131072, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STORAGE(AA, BB, CC, 0)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>TERMTHDACT(UADUMP,,96)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NORTEREUS</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOSIMVRD</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STACK(131072, 131072, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>STORAGE(AA, BB, CC, 0)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>TERMTHDACT(UADUMP,,96)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>HEAP(32768,32768, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NONONIPTSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PAGEFRAMESIZE(4K, 4K, 4K)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOPC</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PLOPCP(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INFOMSGFILTER(OFF,,,)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>IIQFCOPN</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>LIBSTACK(4096,4096,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGFILE(SYSTOUT,FBA,121,0,NOENQ)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGQ(15)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>HEAP(32768,32768, ANYWHERE, KEEP)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOFLOW</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NONONIPTSTACK(See THREADSTACK)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PAGEFRAMESIZE(4K, 4K, 4K)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NOPC</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>PLOPCP(ON)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INFOMSGFILTER(OFF,,,)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>IIQFCOPN</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>INTERRUPT(OFF)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>LIBSTACK(4096,4096,FREE)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGFILE(SYSTOUT,FBA,121,0,NOENQ)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>MSGQ(15)</td>
</tr>
<tr>
<td>IBM-supplied default</td>
<td>NATLANG(ENU)</td>
</tr>
</tbody>
</table>

Using Language Environment debugging facilities 53
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 186.
- For COBOL routines, see “Finding COBOL information in a dump” on page 228.
- For Fortran routines, see “Finding Fortran information in a Language Environment dump” on page 252.
- For PL/I routines, see “Finding PL/I for MVS & VM information in a dump” on page 270.
Table 19: Contents of the Language Environment dump.

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

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

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

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

[5] Enclave Identifier | Names the enclave for which information in the dump is provided. If multiple enclaves are dumped, 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.

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

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

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7] Traceback (continued)</td>
<td>The second part contains:</td>
</tr>
<tr>
<td></td>
<td>• DSA number: A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback.</td>
</tr>
<tr>
<td></td>
<td>• Stack frame (DSA) address</td>
</tr>
<tr>
<td></td>
<td>• Entry point address</td>
</tr>
<tr>
<td></td>
<td>• Program unit address</td>
</tr>
<tr>
<td></td>
<td>• Program unit offset: The offset of the last instruction to run in the routine. If the offset is a negative number, zero, or a very large positive number, the routine associated with the offset probably did not allocate a save area or could have been called using SVC-assisted linkage. Adding the program unit address to the offset gives the location of the current instruction in the routine. This offset is from the starting address of the routine.</td>
</tr>
<tr>
<td></td>
<td>• Compile Date: Contains the year, month and day in which the routine was compiled.</td>
</tr>
<tr>
<td></td>
<td>• Attributes: The available compilation attributes of the compile unit including:</td>
</tr>
<tr>
<td></td>
<td>• A label identifying the LE-supported language such as COBOL, ENT PL/I, C/C++, and so on.</td>
</tr>
<tr>
<td></td>
<td>• Compilation attributes such as EBCDIC, ASCII, IEEE or hexadecimal floating point (HFP). The compilation attributes will only be displayed if there is enough information available.</td>
</tr>
<tr>
<td></td>
<td>• If the CEEDUMP was created under a POSIX environment, POSIX will be displayed.</td>
</tr>
<tr>
<td></td>
<td>The third part of the traceback, which is also referred to as the &quot;Fully Qualified Names&quot; section, contains the following:</td>
</tr>
<tr>
<td></td>
<td>• DSA number</td>
</tr>
<tr>
<td></td>
<td>• Entry</td>
</tr>
<tr>
<td></td>
<td>• Program unit: Similar to the Program Unit column in part 1 except that the server name and the complete program unit (PU) name will be displayed. A PU name will appear here only if it is available to Language Environment.</td>
</tr>
<tr>
<td></td>
<td>• Load Module: The complete pathname of a load module name residing in an UNIX filename will be displayed here if available. The load module's full pathname will be displayed if the PATH environment variable is set such that the pathname of the load module's directory appears before the current directory (.). For load modules found in data sets, the same output shown in the traceback part 1 will also be displayed here.</td>
</tr>
<tr>
<td></td>
<td>The fourth part of the traceback, which is also referred to as the &quot;Full Service Level&quot; section, contains the following:</td>
</tr>
<tr>
<td></td>
<td>• DSA number</td>
</tr>
<tr>
<td></td>
<td>• Entry</td>
</tr>
<tr>
<td></td>
<td>• Service: The full service level string with max length of 64 bytes will be displayed here.</td>
</tr>
<tr>
<td>Section Number and Heading</td>
<td>Contents</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------</td>
</tr>
</tbody>
</table>
| [8] Condition Information for Active Routines | Displays the following information for all conditions currently active on the call chain:  
  - Statement showing failing routine and stack frame address of routine  
  - Condition information block (CIB) address  
  - Current condition, in the form of a Language Environment message for the condition raised or a Language Environment abend code, if the condition was caused by an abend  
  - Location: For the failing routine, this is the program unit, entry routine, statement number, and offset.  
  - Machine state, which shows:  
    - Instruction length counter (ILC)  
    - Interruption code  
    - Program status word (PSW)  
    - Contents of 64-bit GPRs 0–15. Note that when the high halves of the registers are not known, they are shown as ********.  
    - Storage dump near condition (2 hex-bytes of storage near the PSW)  
    - Storage pointed to by General Purpose Registers  
    - Contents of access registers, if available  
  
This information shows the current values at the time the condition was raised. The high halves of the general registers are dumped, in case they are useful for debugging some applications.  

If the PSW associated with the condition indicates AMODE 24, the register content will be treated as 24-bit address.
Table 19: Contents of the Language Environment dump. (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[9] Parameters, Registers, and Variables for Active Routines</td>
<td>For each active routine, this section shows:</td>
</tr>
<tr>
<td></td>
<td>• Routine name and stack frame address</td>
</tr>
<tr>
<td></td>
<td>• Arguments: For COBOL and Fortran, arguments are shown here rather than with the local variables. For COBOL, arguments are shown as part of local variables. PL/I arguments are not displayed in the Language Environment dump.</td>
</tr>
<tr>
<td></td>
<td>• Saved registers: This lists the contents of GPRs 0–15 at the time the routine transferred control.</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• 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:</td>
</tr>
<tr>
<td></td>
<td>• For COBOL, use TEST(SYM).</td>
</tr>
<tr>
<td></td>
<td>• For C/C++, use TEST.</td>
</tr>
<tr>
<td></td>
<td>• For VS COBOL II, use FDUMP.</td>
</tr>
<tr>
<td></td>
<td>• For COBOL/370, use TEST(SYM).</td>
</tr>
<tr>
<td></td>
<td>• For COBOL for OS/390 &amp; VM, use TEST(SYM).</td>
</tr>
<tr>
<td></td>
<td>• For Enterprise COBOL for z/OS V4R2 and prior releases, use TEST(SYM).</td>
</tr>
<tr>
<td></td>
<td>• For Enterprise COBOL for z/OS V5R1 and later releases, use TEST with any sub options or NOTEST(DWARF).</td>
</tr>
<tr>
<td>Note:</td>
<td>- LOW-VALUES (x'00') is the NUL character in EBCDIC, which is an unprintable character that cannot be displayed properly.</td>
</tr>
<tr>
<td></td>
<td>- A NUL character in a data item in the &quot;Local Variables&quot; section is replaced by a double-quote when displayed in the CEEDUMP.</td>
</tr>
<tr>
<td></td>
<td>• For Fortran, use SDUMP.</td>
</tr>
<tr>
<td></td>
<td>• For PL/I, arguments and variables are not displayed.</td>
</tr>
<tr>
<td>[10] Control Blocks for Active Routines</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>• Stack frame</td>
</tr>
<tr>
<td></td>
<td>• Condition information block</td>
</tr>
<tr>
<td></td>
<td>• Language-specific control blocks</td>
</tr>
<tr>
<td>[11] Storage for Active Routines</td>
<td>Displays local storage for each active routine. The storage is dumped in hexadecimal, with EBCDIC translations on the right side of the page. There can be other information, depending on the language used. For C/C++ routines, this is the stack frame storage. For COBOL programs, this is language-specific information, WORKING-STORAGE, and LOCAL-STORAGE.</td>
</tr>
<tr>
<td>Section Number and Heading</td>
<td>Contents</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>[12] Control Blocks</td>
<td>Lists the contents of the Language Environment common anchor area (CAA), thread synchronization queue element (SQEL), DLL failure data, and dummy stack frame. Other language-specific control blocks can appear in this section. DLL failure data is described in “Using the DLL failure control block” on page 79.</td>
</tr>
<tr>
<td>[13] Enclave variables:</td>
<td>Displays language specific global variables. This section also shows the variable type. Variables are displayed only if the symbol tables are available.</td>
</tr>
<tr>
<td>[14] Enclave Control Blocks</td>
<td>Lists the contents of the Language Environment enclave data block (EDB) and enclave member list (MEML). The information presented may vary depending on which runtime options are set.</td>
</tr>
<tr>
<td></td>
<td>• If the POSIX runtime option is set to ON, this section lists the contents of the mutex and condition variable control blocks, the enclave level latch table, and the thread synchronization trace block and trace table.</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• If the HEAPCHK runtime option is set to ON, this section shows the contents of the HEAPCHK options control block (HCOP) and the HEAPCHK element tables (HCEL). A HEAPCHK element table contains the location and length of all allocated storage elements for a heap in the order that they were allocated.</td>
</tr>
<tr>
<td></td>
<td>• When the call-level suboption of the HEAPCHK runtime option is set, any unfreed storage, which would indicate a storage leak, would be displayed in this area. The traceback could then be used to identify the program which did not free the storage.</td>
</tr>
<tr>
<td></td>
<td>• If the TRACE runtime option is set to ON, this section shows the contents of the Language Environment trace table. Other language-specific control blocks can appear in this section.</td>
</tr>
<tr>
<td>[15] Enclave Storage</td>
<td>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.</td>
</tr>
<tr>
<td>[16] File Status and Attributes</td>
<td>Contains additional information about the file.</td>
</tr>
<tr>
<td>[17] Runtime Options Report</td>
<td>Lists the Language Environment runtime options in effect when the routine was executed.</td>
</tr>
<tr>
<td>[18] Process Control Blocks</td>
<td>Lists the contents for the Language Environment process control block (PCB), process member list (MEML), and if the POSIX runtime option is set to ON, the process level latch table. Other language-specific control blocks can appear in this section.</td>
</tr>
<tr>
<td>[19] Additional Language Specific Information</td>
<td>Displays any additional information not included in other sections. For C/C++, it shows the thread ID of the thread that generated the dump and the settings of the errno and errnojr variables for that thread.</td>
</tr>
</tbody>
</table>
Table 19: Contents of the Language Environment dump. (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[20] CEE3846I CEEDUMP Processing completed.</td>
<td>Identifies the end of the Language Environment dump processing. Similarly, message CEE3845I identifies the start of the dump processing. Message CEE3846I can be used to locate the end of the previous CEEDUMP report when scanning backward in a data set that contains several CEEDUMP reports.</td>
</tr>
</tbody>
</table>

Debugging with specific sections of the Language Environment dump

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

Tracebacks, condition information, and data values section

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

Upward-growing (non-XPLINK) stack frame section

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

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

- R1 is a pointer to parameter list or 0 if no parameter list passed.
- R0, R2–R11 is unreferenced by Language Environment. Caller’s values are passed transparently.
- R12 is the pointer to the CAA if entry to an external routine.
- R13 is the pointer to caller’s stack frame.
- R14 is the return address.
- R15 is the address of the called entry point.

With an optimization level other than 0, C/C++ routines save only the registers used during the running of the current routine. Non-Language Environment RSAs can be in the save area chain. The length of the save area and the saved register contents do not always conform to Language Environment conventions. For a detailed description of stack frames Language Environment storage management, see z/OS Language Environment Programming Guide. Figure 12 on page 63 shows the format of the upward-growing stack frame.

Note: The Member-defined fields are reserved for the specific higher level language.
### Figure 12: Upward-growing (non-XPLINK) stack frame format

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Flags</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>CEEDSBACK - Standard 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>CEEDSASAVE - 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>

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

Figure 13 on page 64 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*. 

Using Language Environment debugging facilities 63
### Common Anchor Area

Each thread is represented by a common anchor area (CAA), which is the central communication area for Language Environment. All thread- and enclave-related resources are anchored, provided for, or can be obtained through the CAA. The CAA is generated during thread initialization and deleted during thread termination. When calling Language Environment-conforming routines, register 12 points to the address of the CAA.

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

Table 20 on page 65 describes the CAA fields. For more information about the CAA and other structures to which it refers (for example, the DLL failure control block, CEEDLLF), see *z/OS Language Environment Vendor Interfaces*.

Figure 13: Downward-growing (XPLINK) stack frame format

<table>
<thead>
<tr>
<th>Low Addresses</th>
<th>Stack Pointer (R4)</th>
<th>Stack Frames for called functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Guard Page (4 KB)</td>
</tr>
<tr>
<td></td>
<td>+2048</td>
<td>Backchain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Environment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Entry Point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Return Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R14</td>
</tr>
<tr>
<td></td>
<td>+2096</td>
<td>Savearea (48 bytes)</td>
</tr>
<tr>
<td></td>
<td>+2104</td>
<td>Reserved (8 bytes)</td>
</tr>
<tr>
<td></td>
<td>+2108</td>
<td>Debug Area (4 bytes)</td>
</tr>
<tr>
<td></td>
<td>+2112</td>
<td>Arg Area Prefix (4 Bytes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Argument Area:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parm 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parm 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Local (automatic) Storage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Saved FPRs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Saved ARs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Saved VRs</td>
</tr>
<tr>
<td>High Addresses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>Bit</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Bit</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Bit</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Char</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>Addr</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>Addr</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>Char</td>
<td>52</td>
</tr>
<tr>
<td>44</td>
<td>Signed</td>
<td>2</td>
</tr>
<tr>
<td>46</td>
<td>Signed</td>
<td>2</td>
</tr>
<tr>
<td>48</td>
<td>Char</td>
<td>44</td>
</tr>
<tr>
<td>74</td>
<td>Addr</td>
<td>4</td>
</tr>
<tr>
<td>78</td>
<td>Char</td>
<td>168</td>
</tr>
<tr>
<td>120</td>
<td>Addr</td>
<td>4</td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>124</td>
<td>Char</td>
<td>56</td>
</tr>
<tr>
<td>15C</td>
<td>Addr s</td>
<td>4</td>
</tr>
<tr>
<td>160</td>
<td>Char</td>
<td>56</td>
</tr>
<tr>
<td>198</td>
<td>Bit (96)</td>
<td>12</td>
</tr>
<tr>
<td>1A4</td>
<td>Addr s</td>
<td>4</td>
</tr>
<tr>
<td>1A8</td>
<td>Char</td>
<td>68</td>
</tr>
<tr>
<td>1A8</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1AC</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1B0</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1B4</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1B8</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1BC</td>
<td>Char</td>
<td>32</td>
</tr>
<tr>
<td>1BC</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1C0</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1C4</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1C8</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1CC</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1D0</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1D4</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1D8</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1DC</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1E0</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1E4</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1E8</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>1EC</td>
<td>Bit (32)</td>
<td>4</td>
</tr>
<tr>
<td>1F0</td>
<td>Char</td>
<td>80</td>
</tr>
<tr>
<td>1F0</td>
<td>Addr s</td>
<td>4</td>
</tr>
<tr>
<td>1F4</td>
<td>Addr s</td>
<td>4</td>
</tr>
<tr>
<td>1F8</td>
<td>Char</td>
<td>8</td>
</tr>
<tr>
<td>200</td>
<td>Addr s</td>
<td>4</td>
</tr>
<tr>
<td>204</td>
<td>Signed</td>
<td>4</td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>208</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>20C</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>210</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>214</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>218</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>21C</td>
<td>Char</td>
<td>3</td>
</tr>
<tr>
<td>21F</td>
<td>Char</td>
<td>1</td>
</tr>
<tr>
<td>220</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>224</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>228</td>
<td>Char</td>
<td>2</td>
</tr>
<tr>
<td>22A</td>
<td>Char</td>
<td>2</td>
</tr>
<tr>
<td>22C</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>230</td>
<td>Signed</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>238</td>
<td>Signed</td>
<td>4</td>
</tr>
<tr>
<td>23C</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>240</td>
<td>Char</td>
<td>24</td>
</tr>
<tr>
<td>258</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>25C</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>260</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>264</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>268</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>26C</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>270</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>274</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>278</td>
<td>Addresss</td>
<td>4</td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>27C</td>
<td>Char</td>
<td>4</td>
</tr>
<tr>
<td>280</td>
<td>Address</td>
<td>4</td>
</tr>
<tr>
<td>284</td>
<td>Address</td>
<td>4</td>
</tr>
<tr>
<td>288</td>
<td>Char</td>
<td>36</td>
</tr>
<tr>
<td>2AC</td>
<td>Bit</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2AD</td>
<td>Bit (8)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2AE</td>
<td>Bit (8)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----</td>
</tr>
</tbody>
</table>
| 2AF        | Bit (8) | 1  | CEEAAFLAG2   | CAA Flag 2, defined as follows:  
0  Bimodal addressing is available.  
1  Vector hardware is available.  
2  Thread terminating.  
3  Initial thread  
4  Library trace is active. The TRACE runtime option was set.  
5  Reserved  
6  CEECAA_ENQ_Wait_Interruptible. Thread is in an enqueue wait.  
7  Reserved |
<p>| 2B0        | Unsign | 1  | CEECAALEVEL  | 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. |
| 2B1        | Bit (8) | 1  | CEECAA_PM    | Image of current program mask. |
| 2B2        | Bit (16) | 2  | CEECAA_INVAR | Field that is at the same fixed offset in 31-bit and 64-bit CAAs |
| 2B3        | Bit    | 1  | Reserved     | |
| 2B4        | Address 4  | 4  | CEECAAGETLS | Address of stack overflow for library routines. |
| 2B8        | Address 4  | 4  | CEECAACELV  | Address of the Language Environment library vector. This field is used to locate dynamically loaded Language Environment routines. |
| 2BC        | Address 4  | 4  | CEECAAGETS  | 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. |
| 2C0        | Address 4  | 4  | CEECAAALBOS | 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. |
| 2C4        | Address 4  | 4  | CEECAALEOS  | 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. |</p>
<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2C8</td>
<td>Address</td>
<td>4</td>
<td>CEECAALNAB</td>
<td>Next available library stack storage byte. This contains the address of the next available byte of storage on the library stack. It is modified when library stack storage is obtained or released.</td>
</tr>
<tr>
<td>2CC</td>
<td>Address</td>
<td>4</td>
<td>CEECAADMC</td>
<td>Language Environment shunt routine address. Its value is initially set to 0 during thread initialization. If it is nonzero, this is the address of a routine used in specialized exception processing.</td>
</tr>
<tr>
<td>2D0</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAACD</td>
<td>Most recent CAASHAB abend code.</td>
</tr>
<tr>
<td>2D0</td>
<td>Signed</td>
<td>4</td>
<td>CEEAAABCODE</td>
<td>Most recent abend completion code.</td>
</tr>
<tr>
<td>2D4</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAARS</td>
<td>Most recent CAASHAB reason code.</td>
</tr>
<tr>
<td>2D4</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAARSNCODE</td>
<td>Most recent abend reason code.</td>
</tr>
<tr>
<td>2D8</td>
<td>Address</td>
<td>4</td>
<td>CEECAAERR</td>
<td>Address of the current condition information block. After completion of initialization, this always points to a condition information block. During exception processing, the current condition information block contains information about the current exception being processed. Otherwise, it indicates no exception being processed.</td>
</tr>
<tr>
<td>2DC</td>
<td>Address</td>
<td>4</td>
<td>CEECAAGETSX</td>
<td>Address of the user stack extender routine. This routine is called to extend the current stack frame in the user stack. Its address is in the CEECAA for performance reasons.</td>
</tr>
<tr>
<td>2E0</td>
<td>Address</td>
<td>4</td>
<td>CEECAADDSA</td>
<td>Address of the Language Environment dummy DSA. This address determines whether a stack frame is the dummy DSA, also known as the zeroth DSA.</td>
</tr>
<tr>
<td>2E4</td>
<td>Signed</td>
<td>4</td>
<td>CEECAASECTSIZ</td>
<td>Vector section size. This field is used by the vector math services.</td>
</tr>
<tr>
<td>2E8</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAPARTSUM</td>
<td>Vector partial sum number. This field is used by the vector math services.</td>
</tr>
<tr>
<td>2EC</td>
<td>Signed</td>
<td>4</td>
<td>CEECAASSEXPNT</td>
<td>Log of the vector section size. This field is used by the vector math services.</td>
</tr>
<tr>
<td>2F0</td>
<td>Address</td>
<td>4</td>
<td>CEECEAEDB</td>
<td>Address of the Language Environment EDB. This field points to the encompassing EDB.</td>
</tr>
<tr>
<td>2F4</td>
<td>Address</td>
<td>4</td>
<td>CEECAAPCB</td>
<td>Address of the Language Environment PCB. This field points to the encompassing PCB.</td>
</tr>
<tr>
<td>2F8</td>
<td>Address</td>
<td>4</td>
<td>CEECEAAEYEPT</td>
<td>Address of the CAA eye catcher. The CAA eye catcher is CEECAA. This field can be used for validation of the CAA.</td>
</tr>
<tr>
<td>2FC</td>
<td>Address</td>
<td>4</td>
<td>CEECEAAPTR</td>
<td>Address of the CAA. This field points to the CAA itself and can be used in validation of the CAA.</td>
</tr>
<tr>
<td>300</td>
<td>Address</td>
<td>4</td>
<td>CEECAAGETS1</td>
<td>Non-DSA stack overflow. This field is the address of a stack overflow routine, which cannot guarantee that the current register 13 is pointing at a stack frame. Register 13 must point, at a minimum, to a save area.</td>
</tr>
<tr>
<td>304</td>
<td>Address</td>
<td>4</td>
<td>CEECAASHAB</td>
<td>ABEND shunt routine. Its value is initially set to zero during thread initialization. If it is nonzero, this is the address of a routine used in specialized exception processing for ABENDs that are intercepted in the ESTAE exit.</td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
<td>CAA Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----</td>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>308</td>
<td>Addreses</td>
<td>4</td>
<td>CEECAAPRGCK</td>
<td>Routine interrupt code for CEECAADMC. If CEECAADMC is nonzero, and a routine interrupt occurs, this field is set to the routine interrupt code and control is passed to the address in CEECAAMDC.</td>
</tr>
<tr>
<td>30C</td>
<td>Bit (8)</td>
<td>1</td>
<td>CEECAAFLAG1</td>
<td>CAA flag bits, defined as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 CEECAASORT. A call to DFSORT is active.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 CEECAA_USE_OLD_STK. Use the old stack.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 CEECA_CICS_EXT_REG. ERTLI CICS extended register interface is in effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 CEECAHASHAB_RECOVER_IN_ESTAE_MODE. When on, the Language Environment ESTAE resumes to the abend shunt in the mode and key in which the Language Environment ESTAE was established</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 - 5 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6 CEECA_CICS_VR_SPT. ERTLI CICS vector register interface is in effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 Reserved</td>
</tr>
<tr>
<td>30D</td>
<td>Char</td>
<td>1</td>
<td>CEECAASHAB_KEY</td>
<td>IPK result when CEECAASHAB is set.</td>
</tr>
<tr>
<td>30E</td>
<td>Char</td>
<td>2</td>
<td>Reserved</td>
<td>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>310</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAURC</td>
<td>Determine if a stack overflow routine must be called when allocating storage from the user stack. Normally, the value of this field will represent the end of the current user stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the user stack. This field is used by function prologs that use FASTLINK linkage conventions.</td>
</tr>
<tr>
<td>314</td>
<td>Addreses</td>
<td>4</td>
<td>CEECAAES</td>
<td>Determine if a stack overflow routine must be called when allocating storage from the library stack. Normally, the value of this field will represent the end of the current library stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the library stack. This field is used by function prologs that use FASTLINK linkage conventions.</td>
</tr>
<tr>
<td>318</td>
<td>Addreses</td>
<td>4</td>
<td>CEECAALESS</td>
<td>Determine if a stack overflow routine must be called when allocating storage from the library stack. Normally, the value of this field will represent the end of the current library stack segment. However, its value can also be zero to force the call of a stack overflow routine for every allocation of storage from the library stack. This field is used by function prologs that use FASTLINK linkage conventions.</td>
</tr>
<tr>
<td>31C</td>
<td>Addreses</td>
<td>4</td>
<td>CEECAOGETS</td>
<td>Overflow from user stack allocations.</td>
</tr>
<tr>
<td>320</td>
<td>Addreses</td>
<td>4</td>
<td>CEECAOGETLS</td>
<td>Overflow from library stack allocations.</td>
</tr>
<tr>
<td>Hex Offset</td>
<td>Type</td>
<td>Len</td>
<td>CAA Field</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>-----</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>324</td>
<td>Address</td>
<td>4</td>
<td>CEECAAPICICB</td>
<td>Address of the preinitialization compatibility control block.</td>
</tr>
<tr>
<td>328</td>
<td>Address</td>
<td>4</td>
<td>CEECAAOGETSX</td>
<td>User DSA exit from OPLINK.</td>
</tr>
<tr>
<td>32C</td>
<td>Signed</td>
<td>2</td>
<td>CEECAAGOSMR</td>
<td>Go some more—Used CEEHTRAV multiple.</td>
</tr>
<tr>
<td>32E</td>
<td>Signed</td>
<td>2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>330</td>
<td>Address</td>
<td>4</td>
<td>CEECAALEOV</td>
<td>This field is the address of the Language Environment library vector for z/OS UNIX support.</td>
</tr>
<tr>
<td>334</td>
<td>Signed</td>
<td>4</td>
<td>CEECA_SIGSCTR</td>
<td>SIGSAFE counter.</td>
</tr>
</tbody>
</table>
### Table 20: Description of CAA fields (continued)

<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>338</td>
<td>Bit (32)</td>
<td>4</td>
<td>CEECAA_SIGSFLG</td>
<td>SIGSAFE flags indicate the signal safety of the library and are defined, as follows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 CEECAA_SIGPUTBACK. The signal cannot be delivered, therefore the signal is put back to the kernel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 CEECAA_SA_RESTART. Indicates that a signal registered with the SA_RESTART flag interrupted the last kernel call, and the signal catcher returned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 CEECAA_SIGSAFE. It is safe to deliver the signal, while in library code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 CEECAA_CANCELSAFE. It is safe to deliver the cancel signal, while in library code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5 CEECAA_SIGRESYNCH. CEECAA_sigputsynch flag was on last time CEEOSIGR resolicited a signal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6 CEECAA_FRZ_UNSAFE. This thread is in an unsafe state to be frozen.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 CEECAA_NOAPPREGS. User application registers may be saved in a nonstandard place.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8 CEECAA_EINTR_RSOL. Secondary Signal resolicitation is in progress, after EINTR errno from inner function.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9 CEECAA_EINTR_PUTB. Secondary resolicited signal has been put back.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 CEECAA_EINTR_REST. User signal catcher returned after catching secondary resolicited signal with SA_RESTART in effect.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11 CEECAA_EINTR_SIGG. Stray signal interrupted CEEOSIGG while secondary signal resolicitation was in progress.</td>
</tr>
<tr>
<td>33A</td>
<td>Bit (16)</td>
<td>2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>33C</td>
<td>Char</td>
<td>8</td>
<td>CEECAATHDID</td>
<td>This field is the thread identifier</td>
</tr>
<tr>
<td>344</td>
<td>Address 4</td>
<td></td>
<td>CEECAA_DCRENT</td>
<td>Reserved</td>
</tr>
<tr>
<td>34B</td>
<td>Address 4</td>
<td></td>
<td>CEECAA_DANCHOR</td>
<td>Reserved</td>
</tr>
<tr>
<td>34C</td>
<td>Address 4</td>
<td></td>
<td>CEECAA_CTOC</td>
<td>TOC anchor for CREAT.</td>
</tr>
</tbody>
</table>
Table 20: Description of CAA fields (continued)

<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>354</td>
<td>Signed</td>
<td>4</td>
<td>CEECAACICRSN</td>
<td>CICS reason code from member language.</td>
</tr>
<tr>
<td>358</td>
<td>Address</td>
<td>4</td>
<td>CEECAAMEMBR</td>
<td>Address of thread-level member list.</td>
</tr>
<tr>
<td>35C</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_SIGNAL_STATUS</td>
<td>Signal status of the terminating thread member list.</td>
</tr>
<tr>
<td>360</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_HCOM_REG7</td>
<td>HCOM saved R7.</td>
</tr>
<tr>
<td>360</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_HCOM_REG14</td>
<td>HCOM saved R14.</td>
</tr>
<tr>
<td>364</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_STACKFLOOR</td>
<td>Lowest usable address in XP stack.</td>
</tr>
<tr>
<td>368</td>
<td>Address</td>
<td>4</td>
<td>CEECAAHPGETS</td>
<td>XP stack extension rtn.</td>
</tr>
<tr>
<td>36C</td>
<td>Address</td>
<td>4</td>
<td>CEECAAEDCHPXV</td>
<td>C/C++ XPLINK libvec.</td>
</tr>
<tr>
<td>370</td>
<td>Address</td>
<td>4</td>
<td>CEECAAFOR1</td>
<td>Reserved for FORTRAN.</td>
</tr>
<tr>
<td>374</td>
<td>Address</td>
<td>4</td>
<td>CEECAAFOR2</td>
<td>Reserved for FORTRAN.</td>
</tr>
<tr>
<td>378</td>
<td>Address</td>
<td>4</td>
<td>CEECAATHREADHEAPID</td>
<td>Thread heap ID.</td>
</tr>
<tr>
<td>37C</td>
<td>Signed</td>
<td>4</td>
<td>CEECAA_SYS_RTNCODE</td>
<td>System (kernel) return code.</td>
</tr>
<tr>
<td>380</td>
<td>Signed</td>
<td>4</td>
<td>CEECAA_SYS_RSNCODE</td>
<td>System (kernel) reason code.</td>
</tr>
<tr>
<td>384</td>
<td>Address</td>
<td>4</td>
<td>CEECAAGETFN</td>
<td>Address of the WSA swap routine.</td>
</tr>
<tr>
<td>388</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_JIT1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>38C</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_JIT2</td>
<td>Reserved.</td>
</tr>
<tr>
<td>390</td>
<td>Address</td>
<td>4</td>
<td>CEECAASIGNGPTR</td>
<td>Pointer to 'signam' external variable in a C application.</td>
</tr>
<tr>
<td>394</td>
<td>Signed</td>
<td>4</td>
<td>CEECAASIGNG</td>
<td>Value of sign of lgamma() -1 - negative sign 0 - zero +1 - positive sign.</td>
</tr>
<tr>
<td>398</td>
<td>Address</td>
<td>4</td>
<td>CEECAA_FORDBG</td>
<td>Ptr to AFHDBHIM - FORTRAN hook interface.</td>
</tr>
<tr>
<td>39C</td>
<td>Bit (8)</td>
<td>1</td>
<td>CEECAAAB_STATUS</td>
<td>Validity flags.</td>
</tr>
<tr>
<td>39D</td>
<td>Unsign</td>
<td>1</td>
<td>CEECAA_STACKDIRECTION</td>
<td>Stack direction.</td>
</tr>
<tr>
<td>39E</td>
<td>Bit</td>
<td>2</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3A0</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAAB_GR0</td>
<td>Reg 0 at the time of abend.</td>
</tr>
<tr>
<td>3A4</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAAB_ICD1</td>
<td>SDWAIICD1.</td>
</tr>
<tr>
<td>3A8</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAAB_ABCC</td>
<td>SDWAABCC.</td>
</tr>
<tr>
<td>3AC</td>
<td>Signed</td>
<td>4</td>
<td>CEECAAAB_CRC</td>
<td>SDWACRC.</td>
</tr>
</tbody>
</table>
### Table 20: Description of CAA fields (continued)

<table>
<thead>
<tr>
<th>Hex Offset</th>
<th>Type</th>
<th>Len</th>
<th>CAA Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3B0</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAAGTS</td>
<td>Entry point of CEEVAGTS routine.</td>
</tr>
<tr>
<td>3B4</td>
<td>Address s</td>
<td>4</td>
<td>CEEAA_LERSN1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>3B8</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAHERP</td>
<td>Address of CEEHERP routine.</td>
</tr>
<tr>
<td>3BC</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAUSTKPOS</td>
<td>Start of user stack segment.</td>
</tr>
<tr>
<td>3C0</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAUSTKEOS</td>
<td>End of user stack segment.</td>
</tr>
<tr>
<td>3C4</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAUSERRTN@</td>
<td>Address of thread start routine. Undefined on IPT or prior to thread init event.</td>
</tr>
<tr>
<td>3C8</td>
<td>Bit</td>
<td>8</td>
<td>CEEAAUDHOOK</td>
<td>Hook swapping XPLINK.</td>
</tr>
<tr>
<td>3D0</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAACEL_HPXV_B</td>
<td>Address of XPLINK compat vector for Base library.</td>
</tr>
<tr>
<td>3D4</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAACEL_HPXV_M</td>
<td>Address of XPLINK compat vector for Math library.</td>
</tr>
<tr>
<td>3D8</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAACEL_HPXV_L</td>
<td>Address of XPLINK compat vector for Locale library.</td>
</tr>
<tr>
<td>3DC</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAACEL_HPXV_O</td>
<td>Address of XPLINK compat vector for Open library.</td>
</tr>
<tr>
<td>3E0</td>
<td>Address s</td>
<td>4</td>
<td>CEEAAACEL4VEC3</td>
<td>Address of 3rd C-RTL library vector.</td>
</tr>
<tr>
<td>3E4</td>
<td>Address s</td>
<td>4</td>
<td>CEECA_CEEDLLF</td>
<td>Address of the newest CEEDLLF control block.</td>
</tr>
<tr>
<td>3E8</td>
<td>Address s</td>
<td>4</td>
<td>CEECA_SAVSTACK</td>
<td>Zero or saved stack pointer. This field can be used to save the stack pointer before calling a routine with OS_NOSTACK linkage. After the call returns, this field must be set back to zero.</td>
</tr>
<tr>
<td>3EC</td>
<td>Char</td>
<td>4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3F0</td>
<td>Char</td>
<td>4</td>
<td>CEECA_USER_WORD</td>
<td>4-byte user field available for application use. In pre-initialization (CEEPAPI) environments, this field is initialized in the IPT CAA from the CEEPAPI set_user_word function. This field is initialized to 0 in non-CEEPAPI environments (including all nested enclaves), and for all non-IPT CAAs in CEEPAPI environments. This field is not otherwise accessed by Language Environment.</td>
</tr>
<tr>
<td>3F4</td>
<td>Address s</td>
<td>4</td>
<td>CEECA_SAVSTACK_ASYNC</td>
<td>Zero or address of field that is zero or saved stack pointer. An application that has large sections of code that do not require access to the Language Environment stack but could benefit from having an additional register available can use this field.</td>
</tr>
<tr>
<td>3F8</td>
<td>Char</td>
<td>4</td>
<td>CEECA_STACK_GUARD</td>
<td>Zero or Stack Guard token.</td>
</tr>
</tbody>
</table>

### Condition information block

Figure 14 on page 76 shows the condition information block. The Language Environment condition manager creates a condition information block (CIB) for each condition that is encountered in the...
Language Environment environment. The CIB holds data required by the condition handling facilities and pointers to locations of other data. The address of the current CIB is in the CAA.

For COBOL, Fortran, and PL/I applications, Language Environment provides macros (in the SCEESAMP data set) that map the CIB. For C/C++ applications, the macros are in `leawi.h`.

---

### Figure 14: Condition information block (Part A)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>Condition Information Block Eye catcher</td>
</tr>
<tr>
<td>+4</td>
<td>Previous Condition Information Block</td>
</tr>
<tr>
<td>+8</td>
<td>Most Recent Condition Information Block</td>
</tr>
<tr>
<td>+C</td>
<td>Size of Condition Information Block</td>
</tr>
<tr>
<td></td>
<td>Version of Condition Information Block</td>
</tr>
<tr>
<td>+10</td>
<td>Platform Identifier</td>
</tr>
<tr>
<td></td>
<td>3 = Language Environment</td>
</tr>
<tr>
<td>+18</td>
<td>Current Language Environment Condition</td>
</tr>
<tr>
<td>+24</td>
<td>Address of Machine State</td>
</tr>
<tr>
<td></td>
<td>Time of Interrupt</td>
</tr>
<tr>
<td>+28</td>
<td>Previous Language Environment Condition</td>
</tr>
<tr>
<td>+37</td>
<td>Condition Flags</td>
</tr>
<tr>
<td>+38</td>
<td>Handle Cursor</td>
</tr>
<tr>
<td>+44</td>
<td>Resume Cursor</td>
</tr>
<tr>
<td>+54</td>
<td>Physical Callee Stack Frame Pointer (handle cursor)</td>
</tr>
<tr>
<td>+58</td>
<td>DSA format for Stack frame in the Handle Cursor</td>
</tr>
<tr>
<td></td>
<td>0 = non-XPLINK 1 = XPLINK</td>
</tr>
<tr>
<td>+59</td>
<td>DSA format for Physical Callee/Stack frame (handle cursor)</td>
</tr>
<tr>
<td>+5A</td>
<td>(reserved)</td>
</tr>
<tr>
<td>+5B</td>
<td>(reserved)</td>
</tr>
<tr>
<td>+68</td>
<td>Address of recorded dump dataset name</td>
</tr>
<tr>
<td></td>
<td>(reserved)</td>
</tr>
<tr>
<td>+B0</td>
<td>Status Flag 5</td>
</tr>
<tr>
<td></td>
<td>(64 = An SDWA is Associated with the Condition)</td>
</tr>
<tr>
<td>+B1</td>
<td>Status Flag 6</td>
</tr>
<tr>
<td></td>
<td>(128 = Storage Condition)</td>
</tr>
<tr>
<td>+B2</td>
<td>Status Flag 7</td>
</tr>
<tr>
<td>+B4</td>
<td>Abend Code Word</td>
</tr>
</tbody>
</table>

---
The flags for Condition Flag 4:

2  The resume cursor was moved.

4  The message service processed the condition.

8  The resume cursor was explicitly moved.

The flags for Status Flag 5, Language Environment events:

1  Caused by an attention interrupt.

2  Caused by a signaled condition.

4  Caused by a promoted condition.

8  Caused by a condition management raised TIU.
Caused by a condition signaled via CEEOKILL. 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.)

Caused by a program check.

Caused by an abend.

The flags for Status Flag 6, Language Environment actions:

- **2**: Doing stack frame zero scan.
- **4**: H-cursor pointing to owning SF.
- **8**: Enable only pass (no condition pass).
- **16**: MRC type 1.
- **32**: Resume allowed.
- **64**: Math service condition.
- **128**: Abend reason code valid.

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

The language-specific function codes for the CIB:

- **X'1'**: For condition procedure.
- **X'2'**: For enablement.
- **X'3'**: For stack frame zero conditions.

**Using the machine state information block**

The Language Environment machine state information block contains condition information pertaining to the hardware state at the time of the error. Figure 16 on page 79 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 88 for the contents of CEEDLLF fields.

Multiple enclave dumps

Figure 17 on page 80 illustrates the information available in the Language Environment dump and the order of information for multiple enclaves. If multiple enclaves are used, the dump service generates data and storage information for the most current enclave and moves up the chain of enclaves to the starting enclave in a LIFO order. For example, if two enclaves are used, the dump service first generates output for the most current enclave. Then the service creates output for the previous enclave. A thread terminating in a non-POSIX environment is analogous to an enclave terminating because Language Environment Version 1 supports only single threads.
If multiple nested CEEPIPI Main-DP environments are present, the dump service generates data and storage information for the most current Main-DP environment and moves up the chain of Main-DP environments to the starting Main-DP environment in LIFO order.

When multiple nested CEEPIPI Main-DP environments are present in the dump output, the information in Figure 17 on page 80 appears for the most current Main-DP environment. For the other chained Main-DP environments, only the traceback section appears. The following is an example:

```
**** Information for CEEPIPI token xxxxxxxx ****
information for newest enclave
information for next older enclave
information for oldest enclave
Other information

**** Information for CEEPIPI token xxxxxxxx ****
traceback for newest enclave
```
Generating a system dump

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

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

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

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

You can use these runtime options, with TRAP(ON), to generate a system dump if an unhandled condition of severity 2 or greater occurs. For more details about the level of dump information produced by each of the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 36.

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

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

**ABPERC** *(abcode)*

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

**Abend Codes in Initialization Assembler User Exit**

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

**CEE3ABD**

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

See system or subsystem documentation for detailed system dump information.

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

For details on setting Language Environment runtime options, see *z/OS Language Environment Programming Guide*.

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

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

1. Specify runtime options TERMTHDACT(UAONLY, UADUMP, UATRACE, or UAIMM), and TRAP(ON). If you specify the suboption UAIMM then you must set TRAP(ON,NOSPIE). The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details on the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 36.
2. Decide whether to include a SYSMDUMP DD card or use the DYNDUMP runtime option.
   - Include a SYSMDUMP DD card with the desired data set name and DCB information:
     
     | LRECL=4160, BLKSIZE=4160, and RECFM=FBS. |
   
   - Specify the DYNDUMP runtime option with the following information:
     
     | DYNDUMP (hlq,DYNAMIC,TDUMP) |
   
3. Rerun the program.

Steps for generating a system dump in an IMS runtime environment

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

1. Specify runtime options TERMTHDACT(UAONLY, UADUMP, UATRACE, or UAIMM), ABTERM(ABEND), and TRAP(ON). If you specify the suboption UAIMM, then you must set TRAP(ON,NOSPIE). The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details on the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 36.

2. Decide whether to include a SYSMDUMP DD card or use the DYNDUMP runtime option.
   - Include a SYSMDUMP DD card with the desired data set name and DCB information:
     
     | LRECL=4160, BLKSIZE=4160, and RECFM=FBS. |
   
   - Specify the DYNDUMP runtime option with the following information:
     
     | DYNDUMP (hlq,DYNAMIC,TDUMP) |

3. Rerun the program.

Steps for generating a system dump in a CICS runtime environment

Before you begin: Under CICS, a system dump provides the most useful information for diagnosing problems. However, if you have a Language Environment U4038 abend, CICS will not generate a system dump. To generate diagnostic information for a CICS runtime environment with a Language Environment U4038 abend, you must create a Language Environment U4039 abend. For instructions on how to create a Language Environment U4039 abend, see “Steps for generating a Language Environment U4039 abend” on page 83.

Note: DYNDUMP is ignored in a CICS environment.

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

1. Specify runtime options TERMTHDACT(UAONLY, UADUMP, or UATRACE), ABTERM(ABEND), and TRAP(ON). The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For more details on the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 36.

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

<table>
<thead>
<tr>
<th>STATUS: RESULTS - OVERTYPE TO MODIFY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trd(4088)  Sys    Loc Max( 999 )  Cur(0000)</td>
</tr>
</tbody>
</table>

3. Rerun the program.

Steps for generating a Language Environment U4039 abend

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

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

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

Perform the following steps to generate a system dump from a z/OS UNIX shell:

- Using _BPXK_MDUMP

  1. Specify where to write the system dump

     To write the system dump to a z/OS data set, issue the following command, where filename is a fully qualified data set name with DCB information: LRECL=4160, BLKSIZE=4160, and RECFM=FBS.

        ```
        export _BPXK_MDUMP=filename
        ```

        **Example**
        ```
        export _BPXK_MDUMP=hlq.mydump
        ```

     To write the system dump to an HFS file, issue the following command, where filename is a fully qualified HFS filename.

        ```
        export _BPXK_MDUMP=filename
        ```

        **Example**
        ```
        export _BPXK_MDUMP=/tmp/mydump.dmp
        ```

  2. Specify Language Environment runtime options, where suboption is UAONLY, UADUMP, UATRACE, or UAIMM.

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

        If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For more details about the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 36.

  3. Rerun the program.

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

1. Specify Language Environment runtime options:

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

**suboption**
is UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For more details about the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 36.

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

2. Rerun the program.

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

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

### Formatting and analyzing system dumps

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

### Preparing to use the Language Environment support for IPCS

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

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

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

  ```
  IMBED MEMBER(CEEIPCSP) ENVIRONMENT(IPCS)
  ```

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

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

  **Example**

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

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

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

  ```
  EXIT EP(CEEEEANLZ) ANALYZE
  ```
### Purpose

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

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

### Format

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

Report Type Parameters:
[  SUM  ]
[  HEAP  |  STACK  | SM  ]
[  HPT(number)  |  HPTCB (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

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

### Report type parameters

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

**SUMmary**

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

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

HEAP | STACK | SM

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

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

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

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

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

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

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

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

31
Display entries that are located in virtual storage below the bar.

64
Display entries that are located in virtual storage above the bar.

ALL
Display entries that are located in virtual storage below or above the bar.

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

CM
Requests a report on Condition Management control blocks.

MH
Requests a report on Message Handler control blocks.
CEEdump
Requests a CEEDUMP-like report. The report includes the traceback, the Language Environment trace, and thread synchronization control blocks at process, enclave, and thread levels.

If the dump output has multiple nested enclaves or multiple nested CEEPIPI Main-DP environments, tracebacks will appear for each enclave in each Main-DP environment. This is similar to how the tracebacks appear in the CEEDUMP output. See the section “Multiple enclave dumps” on page 79 for a description of CEEDUMP output when multiple enclave and Main-DP environments are present.

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

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

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

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

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

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

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

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

COBOL
Requests a report on COBOL-specific control blocks.

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

ALL
Requests a report on all the preceding control blocks.

If the value specified in COMP is not one of the values (C, CIO, COBOL, PL/I, 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 reports, as well as C/C++, COBOL, and PL/I reports.

Data selection parameters
Data selection parameters limit the scope of the data in the report. If no data selection parameter is selected, the default is DETAIL.

DETall
Requests formatting all control blocks for the selected components. Only significant fields in each control block are formatted. For the Heap and Storage Management Reports, the DETAIL parameter will provide a detailed heap segment report for each heap segment in the dump. The detailed heap segment report includes information on the free storage tree in the heap segments, and all allocated storage elements. This report will also identify problems that are 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.

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

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

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

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

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

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

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

Examples
For examples of the output that is produced by LEDATA and explanation of the content, refer to “Understanding the Language Environment IPCS VERBEXIT LEDATA output” on page 88.

Understanding the Language Environment IPCS VERBEXIT LEDATA output
The Language Environment IPCS VERBEXIT LEDATA generates formatted output of the Language Environment runtime environment control blocks from a system dump. The following example illustrates the output produced when the LEDATA VERBEXIT is invoked with the ALL parameter. (Ellipses are used to summarize some sections of the dump.) The system dump being formatted was obtained by specifying the TERMTTHDACT(UADUMP) runtime option when running the program CELSAMP in Figure 8 on page 41. “Sections of the Language Environment LEDATA VERBEXIT formatted output” on page 103 describes the information contained in the formatted output. For reference, the sections of the sample dump are numbered to correspond with the descriptions of the formatted output.
Figure 18: Example of formatted output from LEDATA VERBEXIT (Part 1 of 16)

Figure 19: Example of formatted output from LEDATA VERBEXIT (Part 2 of 16)
Figure 20: Example of formatted output from LEDATA VERBEXIT (Part 3 of 16)
Figure 21: Example of formatted output from LEDATA VERBEXIT (Part 4 of 16)
Figure 22: Example of formatted output from LEDATA VERBEXIT (Part 5 of 16)
Free Storage Tree for Heap Segment 21D91018

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Length</th>
<th>Node</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21D96260</td>
<td>00002DB8</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Map of Heap Segment 21D91018

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

Summary of analysis for Heap Segment 21D91018:

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

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

00000000 bytes of storage were not accounted for.

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: 21615D20 +000000 EYE_CATCHER:HPCB FIRST:21D6D000 LAST:22177000 |
| HPSB: 21615E04 +00000000 EYE_CATCHER:HPSB FIRST:21D6D000 LAST:22177000 |
| HANC: 21D6D000 +000000 EYE_CATCHER:HANC NEXT:21D9A000 PREV:21615D20 |
| Free Storage Tree for Heap Segment 21D6D0000

<table>
<thead>
<tr>
<th>Depth</th>
<th>Address</th>
<th>Length</th>
<th>Node</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21D70620</td>
<td>000009E0</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Figure 23: Example of formatted output from LEDATA VERBEXIT (Part 6 of 16)
Figure 24: Example of formatted output from LEDATA VERBEXIT (Part 7 of 16)
Heap Pool Report

+000000 EYE_CATCHER:PCB: LENGTH:00000800 NUMPOOLS:00000006
+0000DC LARGEST_CELL_SIZE:00000800 BIG_REQUESTS:00000001
+000014 STORAGE_HITS_ADDR:21F08028 FLAGS:E400 NUMGETARRAYS:00
+000018 NUMCELLSIZE:06 GET_POOLINFO_ARRAYS_PTR:21D6DA28

Data for pool 1:

+000000 POOL_INDEX:00000001 INPUT_CELL_SIZE:00000008
+000008 CELL_SIZE:00000010 INPUT_PERCENT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000000
+000018 POOL_LATCH_ADDR:21E48B04 POOL_EXTENTS:00000000
+000020 LAST_CELL:21DF6250 NEXT_CELL:21DF55C0
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:00000000
+000030 POOL_NUM_GET_TOTAL:00000002 POOL_NUM_FREE:00000000
+000038 POOL_EXTENTS_ANCHOR:21D95598 POOL_INDEX_SAME_SIZE:01
+00003D POOL_INDEX_SIZE:02 POOL_NUM_SAME_SIZE:01
+000040 POOL_TRACE_TABLE:21EDF070

Heap Pool Extent Mapping

EXTENT: 21D95598
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000

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

21D95540: Allocated storage cell. To display: IP LIST 21D95540 :LEN(X'00000010') ASID(X'01A9')
21D95580: Allocated storage cell. To display: IP LIST 21D95580 :LEN(X'00000010') ASID(X'01A9')

Summary of analysis for Pool 1:
Number of cells: Unused: 202 Free: 0 Allocated: 2 Total Used: 204
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 2:

+000000 POOL_INDEX:00000002 INPUT_CELL_SIZE:00000020
+000008 CELL_SIZE:00000028 INPUT_PERCENT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000005
+000018 POOL_LATCH_ADDR:21E48B08 POOL_EXTENTS:00000000
+000020 LAST_CELL:00000000 NEXT_CELL:00000000
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:00000000
+000030 POOL_NUM_GET_TOTAL:00000000 POOL_NUM_FREE:00000000
+000038 POOL_EXTENTS_ANCHOR:00000000 POOL_INDEX_SAME_SIZE:01
+00003D POOL_INDEX_SIZE:02 POOL_NUM_SAME_SIZE:01
+000040 POOL_TRACE_TABLE:21EDF070

There are no extents for this pool.

Data for pool 6:

+000000 POOL_INDEX:00000006 INPUT_CELL_SIZE:00000800
+000008 CELL_SIZE:00000808 INPUT_PERCENT:00000000
+000010 CELL_POOL_SIZE:00000000 CELL_POOL_NUM:00000004
+000018 POOL_LATCH_ADDR:21E48B08 POOL_EXTENTS:00000000
+000020 LAST_CELL:21D94D80 NEXT_CELL:21D93D70
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:00000000
+000030 POOL_NUM_GET_TOTAL:00000001 POOL_NUM_FREE:00000000
+000038 POOL_EXTENTS_ANCHOR:21D93560 POOL_INDEX_SAME_SIZE:01
+00003D POOL_INDEX_SIZE:06 POOL_NUM_SAME_SIZE:01
+000040 POOL_TRACE_TABLE:21F9F0F0

Heap Pool Extent Mapping

EXTENT: 21D93560
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000

To display entire pool extent: IP LIST 21D93560 :LEN(X'00002028') ASID(X'01A9')

21D93540: Allocated storage cell. To display: IP LIST 21D93540 :LEN(X'00000808') ASID(X'01A9')
21D93570: Allocated storage cell. To display: IP LIST 21D93570 :LEN(X'00000000') ASID(X'01A9')

Summary of analysis for Pool 6:
Number of cells: Unused: 3 Free: 0 Allocated: 1 Total Used: 4
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Figure 25: Example of formatted output from LEDATA VERBEXIT (Part 8 of 16)
Figure 26: Example of formatted output from LEDATA VERBEXIT (Part 9 of 16)
To display entire DSA: IP LIST 21D71D20 LEN(X'000030F8') ASID(X'01A9')

DSA: 21D71D20
+000000 FLAGS:10CC MEMD:CCC BKC:21D71130 FWC:21D7132B
+00000C R14:A166BA66 R15:21BDA45C R0:216011A4
+000018 R1:21D71760 R2:21D71300 R3:21600ED0
+000024 R4:21D71302 R5:21608E0D R6:21D7130C
+000030 R7:21D71210 R8:00000030 R9:80000000
+000048 LDS:00000000 NAT:21D7132B PNB:CCCCCCCC
+000064 RENT:CCCCCCCC CILC:CCCCCCCC MODE:CCCCCCCC
+000078 RMR:CCCCCCCC

Contents of DSA at location 21D71248:

+00000000 10CCCCCC 21D71130 21D7128 A1600A66 2180A45C 216011A4 21D71260 21D71305 |.....P.....u...P...P...|
+00000020 21D71302 21600ED0 21D71310 20000000 00000000 A1D2E8B2 |.--------.....P...|
+00000040 21D71308 21600ED0 21D71310 20000000 00000000 A1D2E8B2 |.--------.....P...|
+00000060 21D7130C 21600ED0 21D71310 20000000 00000000 A1D2E8B2 |.--------.....P...|
+00000080 21D71310 21600ED0 21D71310 20000000 00000000 A1D2E8B2 |.--------.....P...|
+000000A0 21D71310 21600ED0 21D71310 20000000 00000000 A1D2E8B2 |.--------.....P...|
+000000C0 21D71310 21600ED0 21D71310 20000000 00000000 A1D2E8B2 |.--------.....P...|
+000000E0 21D71310 21600ED0 21D71310 20000000 00000000 A1D2E8B2 |.--------.....P...|
+00000100 21D71248 |.................|

DSA: 21D71130
+000000 FLAGS:10CC MEMD:CCCC BKC:21D71030 FWC:CCCCCCCC
+00000C R14:A169310E R15:21D2E8BE R0:7D000009
+000018 R1:21D710B0 R2:21604328 R3:0000002
+000024 R4:A169303A R5:216157D0 R6:216039EC
+000030 R7:21D710D0 R8:00000030 R9:80000000
+000048 LDS:00000000 NAT:21D71130 PNB:CCCCCCCC
+000064 RENT:CCCCCCCC CILC:CCCCCCCC MODE:CCCCCCCC
+000078 RMR:CCCCCCCC

Contents of DSA at location 21D71130:

+00000000 0000CCCC 21617660 CCCCCCCC A169310E 21D2E8BE 7D000009 21D710B0 21D710D0 |.....-...P...-.Y.KZ.| 21D710D8 21D710D4
+00000020 00000002 A169303A 216157D0 216039EC 21604330 00000030 00000008 A1D2E8B2 |........./...-...-P...|
+00000040 A1692F48 21616BB8 00000000 21D71130 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |...../,......P..................|
+00000060 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+00000080 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000A0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000C0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000E0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|

To display entire DSA: IP LIST 21D71130 LEN(X'00000118') ASID(X'01A9')

DSA: 21D71030
+000000 FLAGS:0000 MEMD:CCC BKC:21617660 FWC:CCCCCCCC
+00000C R14:A1699310E R15:21D2E8B64 R0:70000009
+000018 R1:21D710B0 R2:21604328 R3:00000002
+000024 R4:A1699303A R5:216157D0 R6:216039EC
+000030 R7:21D710D0 R8:00000030 R9:80000000
+000048 LDS:00000000 NAT:21D71130 PNB:CCCCCCCC
+000064 RENT:CCCCCCCC CILC:CCCCCCCC MODE:CCCCCCCC
+000078 RMR:CCCCCCCC

Contents of DSA at location 21D71030:

+00000000 0000CCCC 21617660 CCCCCCCC A1699310E 21D2E8B64 70000009 21D710B0 21D710D0 |.....-...P...-.Y.KZ.| 21D710D8 21D710D4
+00000020 00000002 A1699303A 216157D0 216039EC 21604330 00000030 00000008 A1D2E8B2 |........./...-...-P...|
+00000040 A1692F48 21616BB8 00000000 21D71130 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |...../,......P..................|
+00000060 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+00000080 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000A0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|
+000000C0 CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC CCCCCCCC |................................|

User Stack Control Blocks

STKH: 21D71018
+000000 EYE_CATCHER:STKU NEXT:2161742C PREV:2161742C
+00000C SEGMENT_LEN:00620080

Figure 27: Example of formatted output from LEDATA VERBEXIT (Part 10 of 16)
Figure 28: Example of formatted output from LEDATA VERBEXIT (Part 11 of 16)
Figure 29: Example of formatted output from LEDATA VERBEXIT (Part 12 of 16)
Figure 30: Example of formatted output from LEDATA VERBEXIT (Part 13 of 16)
Message Processing Control Blocks

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

MDST forward chain from CMXBDHEAD(1)

  MOST: 00016000
+000000 EYE:MDST SIZE:0100 CTL:40 CEEDUMPLOC:00
+000008 NEXT:00012000 PREV:00000000 DDNAM:CEEDUMP

  MOST: 00012000
+000000 EYE:MDST SIZE:0100 CTL:40 CEEDUMPLOC:00
+000008 NEXT:00000000 PREV:00016000 DDNAM:SYSOUT

MDST back chain from CMXBDHEAD(2)

  MOST: 00016000
+000000 EYE:MDST SIZE:0100 CTL:40 CEEDUMPLOC:00
+000008 NEXT:00000000 PREV:00016000 DDNAM:CEEDUMP

  MOST: 00012000
+000000 EYE:MDST SIZE:0100 CTL:40 CEEDUMPLOC:00
+000008 NEXT:00016000 PREV:00000000 DDNAM:SYSOUT

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

  MGF: 22167028
+000000 EYE:CMIB PREV:22167028 NEXT:22118380 SEQ:00000005
+000010 CTOK:00000BF7 41C3C5C5 (CEE3063I)

  MGF: 22118380
+000000 EYE:CMIB PREV:22167028 NEXT:22118380 SEQ:00000002
+000010 CTOK:00030C89 59C3C5C5 (CEE3209S)

  MGF: 2160F080
+000000 EYE:CMIB PREV:22118380 NEXT:221315C0 SEQ:00000001
+000010 CTOK:000300FC 41C3C5C5 (CEE3574I)

  MGF: 221315C0
+000000 EYE:CMIB PREV:2160F080 NEXT:22131780 SEQ:00000003
+000010 CTOK:0003001C 59C3C5C5 (CEE0462S)

  MGF: 22131780
+000000 EYE:CMIB PREV:221315C0 NEXT:22167028 SEQ:00000004
+000010 CTOK:00030159 49C3C5C5 (CEE0455W)

Information for enclave main

Registers and PSW:

GPR0..... 00000000_84000000 GPR1..... 00000000_84000FC7 GPR2..... 00000000_21D72618 GPR3..... 00000000_00020009
GPR4..... 00000000_21616E88 GPR5..... 00000000_21616DC2 GPR6..... 00000000_21616D20 GPR7..... 00000000_21608430
GPR8..... 00000000_21D72618 GPR9..... 00000000_21D721AC GPR10..... 00000000_21D7201F GPR11..... 00000000_A169B58
GPR12..... 00000000_21616B88 GPR13..... 00000000_21D74E18 GPR14..... 00000000_A169B8E GPR15..... 00000000_00000000
PSW..... 07BD504B 169C35C2

Figure 31: Example of formatted output from LEDATA VERBEXIT (Part 14 of 16)
Figure 32: Example of formatted output from LEDATA VERBEXIT (Part 15 of 16)
Table 21: Contents of the LEDATA VERBEXIT formatted output.

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] - [9] Summary:</td>
<td>The following sections are included when the SUMMARY parameter is specified on the LEDATA invocation.</td>
</tr>
<tr>
<td>[1] Summary Header</td>
<td>Contains the following information:</td>
</tr>
<tr>
<td></td>
<td>• Address of Thread control block (TCB)</td>
</tr>
<tr>
<td></td>
<td>• Release number</td>
</tr>
<tr>
<td></td>
<td>• Address Space ID (ASID)</td>
</tr>
<tr>
<td>[2] Active Members List</td>
<td>List of active members is extracted from the enclave member list (MEML)</td>
</tr>
<tr>
<td>[3] CECECAA</td>
<td>Formats the contents of the Language Environment common anchor area (CAA). See “Common Anchor Area” on page 64 for a description of the fields in the CAA.</td>
</tr>
<tr>
<td>[4] CEEDLLF</td>
<td>Formats the contents of all Language Environment CEEDLLF (DLLF) control blocks that are in use. See CEEDLLF — DLL failure control block in z/OS Language Environment Vendor Interfaces for more information about the CEEDLLF control block chain.</td>
</tr>
<tr>
<td>[5] CEEPCLB</td>
<td>Formats the contents of the Language Environment process control block (PCB), and the process level member list.</td>
</tr>
<tr>
<td>Section number and heading</td>
<td>Contents</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>[7] CEEEDB</td>
<td>Formats the contents of the Language Environment enclave data block (EDB), and the enclave level member list.</td>
</tr>
<tr>
<td>[8] PMCB</td>
<td>Formats the contents of the Language Environment program management control block (PMCB).</td>
</tr>
<tr>
<td>[9] Runtime Options</td>
<td>Lists the runtime options in effect at the time of the dump, and indicates where they were set.</td>
</tr>
</tbody>
</table>
| [10] Heap Storage Control Blocks | This section is included when the HEAP or SM parameter is specified on the LEDATA invocation. It formats the Enclave-level storage management control block (ENSM) and for each different type of heap storage:  
  - Heap control block (HPCB)  
  - Chain of heap anchor blocks (HANC). A HANC immediately precedes each segment of heap storage.  
  This section includes a detailed heap segment report for each segment in the dump. For more information about the detailed heap segment report, see “Understanding the HEAP LEDATA output” on page 107.  
  When HEAPPOOLS is ON, this section also includes a detailed heap pools report. For more information about the detailed heap pools report, see “Understanding the heap pools LEDATA output” on page 111. |
| [11] Stack Storage Control Blocks | This section is included when the STACK or SM parameter is specified on the LEDATA invocation; it formats:  
  - Storage management control block (SMCB)  
  - Chain of dynamic save areas (DSA). See “Upward-growing (non-XPLINK) stack frame section” on page 62 or “Downward-growing (XPLINK) stack frame section” on page 63 for a description of the fields in the DSA.  
  - Chain of stack segment headers (STKH). An STKH immediately precedes each segment of stack storage. |
| [12] Condition Management Control Blocks | This section is included when the CM parameter is specified on the LEDATA invocation; it formats the chain of Condition Information Block Headers (CIBH) and Condition Information Blocks. The Machine State Information Block is contained with the CIBH starting with the field labeled MCH_EYE. See “Condition information block” on page 75 for a description of fields in these control blocks. |
| [13] Message Processing Control Blocks | This section is included when the MH parameter is specified on the LEDATA invocation. |
| [14]-[17] NTHREADS information: | One or more instances of these sections are included when the NTHREADS() parameter is specified on the LEDATA invocation. For a description of NTHREADS, see “Report type parameters” on page 85. |
| [14]- [21] CEEDUMP Formatted Control Blocks: | These sections are included when the CEEDUMP parameter is specified on the LEDATA invocation. |
| [14] Enclave Identifier | Names the enclave for which information is provided. |
| [15] Information for thread | Shows the system identifier for the thread. Each thread has a unique identifier. |
| [16] Registers and PSW | 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. |
Table 21: Contents of the LEDATA VERBEXIT formatted output. (continued)

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [17] Traceback            | For all active routines in a particular thread, the traceback section shows routine information in two parts. The first part contains the following items:  
  - DSA number: A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second part of the traceback.  
  - Entry: For COBOL, Fortran, and PL/I routines, this is the entry point name. For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, the string "** NoName **" will appear.  
  - Entry point offset  
  - Statement number: This field contains Language Environment data.  
  - Load module  
  - Program unit: The primary entry point of the external procedure. For COBOL programs, this is the PROGRAM-ID name. For C, Fortran, and PL/I routines, this is the compile unit name. For Language Environment-conforming assemblers, this is the EPNAME = value on the CEEPPA macro.  
  - Service level: The latest service level applied to the compile unit (for example, for IBM products, it would be the PTF number).  
    - If the service level string is equal or less than 7 bytes, all of the string will be output.  
    - If the service level string is longer than 7 bytes, the Service column will only show the first 7 bytes of the service string, and the full service string will be shown in section of Full Service Level with max length of 64 bytes.  
  - Status: Routine status can be call, exception, or running.  

The second part contains the following items:  
  - DSA number: A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second part of the traceback.  
  - Stack frame (DSA) address  
  - Entry point address  
  - Program unit address  
  - Program unit offset: The offset of the last instruction to run in the routine. If the offset is a negative number, zero, or a very large positive number, the routine associated with the offset probably did not allocate a save area, or the routine could have been called using SVC-assisted linkage. Adding the program unit address to the offset gives you the location of the current instruction in the routine. This offset is from the starting address of the routine.  
  - Compile Date: Contains the year, month and day in which the routine was compiled.  
  - Attributes: The available compilation attributes of the compile unit include:  
    - A label identifying the LE-supported language such as COBOL, ENT PL/I, C/C++, and so on.  
    - Compilation attributes such as EBCDIC, ASCII, IEEE, or hexadecimal floating point (HFP). The compilation attributes will only be displayed if there is enough information available.  
    - POSIX, If the CEEDUMP was created under a POSIX environment.  

The third part of the traceback, which is also referred to as the “Full Service Level” section, contains the following:  
  - 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 point address  
  - Service: The full service level string with max length of 64 bytes will be displayed here.  

| [18] Control Blocks Associated with the Thread | Lists the contents of the thread synchronization queue element (SQEL). |
| [19] Enclave Control Blocks | If the POSIX runtime option was set to ON, this section lists the contents of the mutex and condition variable control blocks, the enclave level latch table, and the thread synchronization trace block and trace table. If the HEAPCHK runtime option is set to ON, this section lists the contents of the HEAPCHK options control block (HCOP) and the HEAPCHK element tables (HCHEL). 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 runtime option was set to ON, this section shows the contents of the Language Environment trace table. |
| [21] Process Control Blocks | If the POSIX runtime 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 PIPICB associated with CAA is displayed instead. |

**PTBL LEDATA output**

The VERBEXIT LEDATA command generates formatted output of PreInit tables when the PTBL or ALL parameter are specified. If ALL is specified, PTBL defaults to CURRENT value. The following sample illustrates the output produced when the VERBEXIT LEDATA command is invoked with the PTBL parameter.

```
PTBL(CURRENT)
********************************************************************************
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  : CEEIXIPTB
TCB address : 008D6E88

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
( 20905DB8 )

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

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

CEEPIPI Trace Table
Entries :
Call Type = INIT_MAIN
PIPI Driver Address  = A099068A
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 =
When nested CEEPIPI main-DP environments are present, two new items will appear after the TCB address:

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

The following is an example:

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

Understanding the HEAP LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap segment report when the HEAP option is used with the DETAIL option, or when the SM,DETAIL option is specified. The detailed heap segment report is useful when trying to pinpoint damage because it provides very specific information. The report describes the nature of the damage, and specifies where the actual damage occurred. The report can also be used to diagnose storage leaks, and to identify heap fragmentation. The following example illustrates the output produced by specifying the HEAP option. “Heap report sections of the LEDATA output” on page 109 describes the information contained in the formatted output. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows in Table 22 on page 110. 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.
This is the last heap segment in the current heap.

1. Free Storage Tree for Heap Segment 25995000

   Node  Node  Parent  Left  Right  Left  Right
   Depth  Address  Length  Node  Node  Node  Length  Length
   0  25995000  88007f50  00000000  00000000  00000000  00000000

2. Map of Heap Segment 25995000

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

Summary of analysis for Heap Segment 25995000:

Amounts of identified storage: Free: 00000000 Allocated: 00000008 Total: 00000008

Numbers of identified areas: Free: 1 Allocated: 4 Total: 5

No errors were found while processing this heap segment.

Anywhere Heap Control Blocks

HPCB: 00014D78
   +000000  EYE_CATCHER:HANC  NEXT:00014D78  LAST:25995000
   +000000  EYE_CATCHER:HPCB  FIRST:25995000  LAST:25995000

Free Storage Tree for Heap Segment 24A91000

The free storage tree is empty.

Map of Heap Segment 24A91000

To display entire segment: IP LIST 24A91000 LEN('X'00000000') ASID('X'0021')

Summary of analysis for Heap Segment 24A91000:

Amounts of identified storage: Free: 00000000 Allocated: 00F00008 Total: 00F00008

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

No errors were found while processing this heap segment.

Map of Heap Segment 25995000

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

Summary of analysis for Heap Segment 25995000:

Amounts of identified storage: Free: 00000000 Allocated: 00F00008 Total: 00F00008

Numbers of identified areas: Free: 0 Allocated: 5 Total: 5

No errors were found while processing this heap segment.

Anywhere Heap Control Blocks

HPCB: 00014D78
   +000000  EYE_CATCHER:HANC  NEXT:25995000  LAST:25995000
   +000000  EYE_CATCHER:HPCB  FIRST:25995000  LAST:25995000

Free Storage Tree for Heap Segment 25995000

The free storage tree is empty.
Heap report sections of the LEDATA output

The Heap Report sections of the LEDATA output provide information for each heap segment in the dump. The detailed heap segment reports include information on the free storage tree in the heap segments, the allocated storage elements, and the cause of heap management data structure problems.
Table 22: Contents of the Heap report sections of LEDATA output

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [1] Free Storage Tree Report       | Within each heap segment, Language Environment tracks deallocated 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 that describes the problem directly after the formatted line of the node that failed validation. |
| [2] Heap Segment Map Report         | The LEDATA HEAP option produces a report that lists all of the storage areas within each heap segment, and identifies the area as either allocated or freed. For each allocated area the contents of the first X’20 bytes of the area are displayed in order to help identify the reason for the storage allocation. Each allocated storage element has an 8-byte prefix that is used by Language Environment to manage the area. The first fullword contains a pointer to the start of the heap segment. The second fullword contains the length of the allocated storage element. The formatter validates this header to ensure that its heap segment pointer is valid. The length is also validated to ensure that it:  
  • Is a multiple of 8.  
  • Is not zero.  
  • Is not larger than the heap segment length.  
  • Does not cause the end of the element to fall outside of the current heap segment.  
  • Does not cause the element to overlap a free storage node. If the heap_free_value of the STORAGE runtime option was specified, then the formatter also checks that the free storage within each free storage element is set to the requested heap_free_value. If a problem is found, then an error message that describes the problem is placed after the formatted line of the storage element that failed validation. |

**Diagnosing heap damage problems**

Heap storage errors can occur when an application allocates a heap storage element that is too small for it to use, and therefore, accidently overlays heap storage. If this situation occurs then some of the typical error messages generated are:

• The node address does not represent a valid node within the heap segment
• The length of the segment is not valid, or
• The heap segment pointer is not valid.

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

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

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

Diagnosing storage leak problems

A storage leak occurs when a program does not return storage back to the heap after it has finished using it. To determine if this problem exists, do one of the following:

- The call-level suboption of the HEAPCHK runtime option causes a report to be produced in the CEEDUMP. Any still-allocated (that is, not freed) storage identified by HEAPCHK is listed in the report, along with the corresponding traceback. This shows any storage that wasn’t freed, as well as all the calls that were involved in allocating the storage. For more information about the HEAPCHK runtime option, see z/OS Language Environment Programming Reference.

- Examine the Heap Segment Map report to see if any data areas, within the allocated storage elements, appear more frequently than expected. If they do, then check to see if these data areas are still being used by the application program. If the data areas are not being used, then change the program to free the storage element after it is done with it.

Diagnosing heap fragmentation problems

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

Understanding the heap pools LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap pools report when HEAPPPOOLS is ON. The detailed heap pools report is useful when trying to find potential damaged cells because it provides very specific information. The following sample shows an example of a report and “Heap pools report sections of the LEDATA output” on page 115 describes the information contained in the formatted output.
No errors were found while processing free chain.

Summary of analysis for Pool 1:
Number of cells: Unused: 19 Free: 1 Allocated: 0 Total Used: 20
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 2:
POOLDATA: 25C1F00
+000000 POOLDATA:00000002 INPUT_CELL_SIZE:00000020
+000008 CELL_SIZE:00000020 INPUT_PERCENT:00000001
+000010 CELL_POOL_SIZE:00000010 CELL_POOL_NUM:00000008
+000018 POOLDATA:25C54BE8 POOLDATA:00000001
+000020 LAST_CELL:25C45D68 NEXT_CELL:25C45C78
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:25C45C50
+000030 POOLDATA:25C54B8 POOLDATA:00000000
+000038 POOLDATA:25C45C48 POOLDATA:00000001
+00003D POOLDATA:02 POOLDATA:00000001
+000040 POOLDATA:25C86080

EXTENT: 25C45C48
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire pool extent: IP LIST 25C45C48 LEN(X'00000148') ASID(X'0020')
25C45C50: Free storage cell. To display: IP LIST 25C45C50 LEN(X'00000028') ASID(X'0020')

[1] Verifying free chain for pool: 2...
No errors were found while processing free chain.
Summary of analysis for Pool 2:
Number of cells: Unused: 7 Free: 1 Allocated: 0 Total Used: 8
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 3:
POOLDATA: 25C1F000
+000000 POOLDATA:00000003 INPUT_CELL_SIZE:00000080
+000008 CELL_SIZE:00000088 INPUT_PERCENT:00000001
+000010 CELL_POOL_SIZE:00000220 CELL_POOL_NUM:00000004
+000018 POOLDATA:25C54BFC POOLDATA:00000002
+000020 LAST_CELL:25C45F38 NEXT_CELL:25C45F38
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:25C45DA0
+000030 POOLDATA:25C45D98 POOLDATA:00000002
+000038 POOLDATA:25C45D98 POOLDATA:00000001
+00003D POOLDATA:03 POOLDATA:00000001
+000040 POOLDATA:25C860A0

EXTENT: 25C45D98
+000000 EYE_CATCHER:EX31 NEXT_EXTENT:00000000
To display entire pool extent: IP LIST 25C45D98 LEN(X'000000228') ASID(X'0020')
25C45D98: Allocated storage cell. To display: IP LIST 25C45D98 LEN(X'000000088') ASID(X'0020')
25C45D98: Allocated storage cell. To display: IP LIST 25C45D98 LEN(X'000000008') ASID(X'0020')
25C45D98: Allocated storage cell. To display: IP LIST 25C45D98 LEN(X'000000088') ASID(X'0020')
25C45D98: Allocated storage cell. To display: IP LIST 25C45D98 LEN(X'000000088') ASID(X'0020')
25C45D98: Allocated storage cell. To display: IP LIST 25C45D98 LEN(X'000000088') ASID(X'0020')
25C45D98: Allocated storage cell. To display: IP LIST 25C45D98 LEN(X'000000088') ASID(X'0020')
25C45D98: Allocated storage cell. To display: IP LIST 25C45D98 LEN(X'000000088') ASID(X'0020')
25C45D98: Allocated storage cell. To display: IP LIST 25C45D98 LEN(X'000000088') ASID(X'0020')
25C45D98: Allocated storage cell. To display: IP LIST 25C45D98 LEN(X'000000088') ASID(X'0020')

[1] Verifying free chain for pool: 3...
No errors were found while processing free chain.
Summary of analysis for Pool 3:
Number of cells: Unused: 1 Free: 3 Allocated: 4 Total Used: 8
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 4:
POOLDATA: 25C1F000
+000000 POOLDATA:00000004 INPUT_CELL_SIZE:00000010
+000008 CELL_SIZE:00000010 INPUT_PERCENT:00000001
+000010 CELL_POOL_SIZE:00000040 CELL_POOL_NUM:00000004
+000018 POOLDATA:25C54C10 POOLDATA:00000001
+000020 LAST_CELL:25C45DE8 NEXT_CELL:25C45E0
+000028 Q_CONTROL_INFO:00000000 Q_FIRST_CELL:25C45FD0
+000030 POOLDATA:25C45E0 POOLDATA:00000000

z/OS: Language Environment Debugging Guide
### Heap Pool Extent Mapping

**EXTENT:** 25C45FC8

- **EYE_CATCHER:** EX31
- **NEXT_EXTENT:** 00000000

To display entire pool extent: IP LIST 25C45FC8 LEN(X'00000428') ASID(X'0020')

25C45FD0: Free storage cell. To display: IP LIST 25C45FD0 LEN(X'00000108') ASID(X'0020')

25C460D8: Free storage cell. To display: IP LIST 25C460D8 LEN(X'00000108') ASID(X'0020')

---

**Summary of analysis for Pool 4:**

- Number of cells: Unused: 2  Free: 2  Allocated: 0  Total Used: 4
- 00000000 free cells were not accounted for.
- No errors were found while processing free chain.

No errors were found while processing this Pool.

---

**Data for pool 5.1:**

- **POOLDATA:** 25C1F200
- **POOL_INDEX:** 00000005
- **INPUT_CELL_SIZE:** 00000400
- **CELL_SIZE:** 00000408
- **INPUT_PERCENT:** 00000001
- **CELL_POOL_SIZE:** 00001020
- **CELL_POOL_NUM:** 00000004
- **POOL_LATCH_ADDR:** 25C54C24
- **POOL_EXTENTS:** 00000002
- **LAST_CELL:** 25E48C48
- **NEXT_CELL:** 25E48438
- **Q_CONTROL_INFO:** 000001DD
- **Q_FIRST_CELL:** 25C42858
- **POOL_NUM_GET_TOTAL:** 000000F2
- **POOL_NUM_FREE:** 00000003
- **POOL_EXTENTS_ANCHOR:** 25E48028
- **POOL_INDEX_SAME_SIZE:** 01
- **POOL_INDEX_SIZE:** 05
- **POOL_NUM_SAME_SIZE:** 05
- **POOL_TRACE_TABLE:** 25D160E0

---

**EXTENT:** 25E48028

- **EYE_CATCHER:** EX31
- **NEXT_EXTENT:** 25C42040

To display entire pool extent: IP LIST 25E48028 LEN(X'00001028') ASID(X'0020')

25E48030: Free storage cell. To display: IP LIST 25E48030 LEN(X'00000408') ASID(X'0020')

25E48048: Free storage cell. To display: IP LIST 25E48048 LEN(X'00000408') ASID(X'0020')

25E48050: Allocated storage cell. To display: IP LIST 25E48050 LEN(X'00000408') ASID(X'0020')

25E48058: Free storage cell. To display: IP LIST 25E48058 LEN(X'00000408') ASID(X'0020')

---

**Summary of analysis for Pool 5.1:**

- Number of cells: Unused: 3  Free: 3  Allocated: 2  Total Used: 8
- 00000000 free cells were not accounted for.
- No errors were found while processing free chain.

No errors were found while processing this Pool.

---

**Data for pool 5.2:**

- **POOLDATA:** 25C1F300
- **POOL_INDEX:** 00000006
- **INPUT_CELL_SIZE:** 00000400
- **CELL_SIZE:** 00000408
- **INPUT_PERCENT:** 00000001
- **CELL_POOL_SIZE:** 00001020
- **CELL_POOL_NUM:** 00000004
- **POOL_LATCH_ADDR:** 25C54C24
- **POOL_EXTENTS:** 00000001
- **LAST_CELL:** 25C47018
- **NEXT_CELL:** 25C47018
- **Q_CONTROL_INFO:** 000001DD
- **Q_FIRST_CELL:** 25C46400
- **POOL_NUM_GET_TOTAL:** 000000F0
- **POOL_NUM_FREE:** 00000003
- **POOL_EXTENTS_ANCHOR:** 25C463F8
- **POOL_INDEX_SAME_SIZE:** 02
- **POOL_INDEX_SIZE:** 05
- **POOL_NUM_SAME_SIZE:** 05
- **POOL_TRACE_TABLE:** 25D46100

---

**EXTENT:** 25C463F8

- **EYE_CATCHER:** EX31
- **NEXT_EXTENT:** 00000000

To display entire pool extent: IP LIST 25C463F8 LEN(X'00001028') ASID(X'0020')

25C46400: Free storage cell. To display: IP LIST 25C46400 LEN(X'00000408') ASID(X'0020')

25C46808: Free storage cell. To display: IP LIST 25C46808 LEN(X'00000408') ASID(X'0020')

25C46C10: Free storage cell. To display: IP LIST 25C46C10 LEN(X'00000408') ASID(X'0020')

---

**Summary of analysis for Pool 5.2:**

- Number of cells: Unused: 1  Free: 3  Allocated: 0  Total Used: 4
- 00000000 free cells were not accounted for.
- No errors were found while processing free chain.

No errors were found while processing this Pool.

---

**Data for pool 5.3:**

- **POOLDATA:** 25C1F400
- **POOL_INDEX:** 00000007
- **INPUT_CELL_SIZE:** 00000400

---

Using Language Environment debugging facilities 113
### Data for pool 5.3:

**Data for pool 5.3:**

**POOLDATA:** 25C1F500

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOL_INDEX</td>
<td>00000008</td>
</tr>
<tr>
<td>INPUT_CELL_SIZE</td>
<td>000000400</td>
</tr>
<tr>
<td>CELL_INDEX</td>
<td>00000000</td>
</tr>
<tr>
<td>CELL_POOL_SIZE</td>
<td>00001020</td>
</tr>
<tr>
<td>NEXT_CELL</td>
<td>25E49C78</td>
</tr>
<tr>
<td>POOL_EXTENSION</td>
<td>00000001</td>
</tr>
<tr>
<td>INPUT_PERCENT</td>
<td>00000001</td>
</tr>
<tr>
<td>CELL_POOL_INDEX</td>
<td>00000000</td>
</tr>
<tr>
<td>CELL_POOL_NUM</td>
<td>000000000004</td>
</tr>
<tr>
<td>LAST_CELL</td>
<td>25C48048</td>
</tr>
<tr>
<td>POOL_LATCH_ADDR</td>
<td>25C54C24</td>
</tr>
<tr>
<td>Q_CONTROL_INFO</td>
<td>00000001D</td>
</tr>
<tr>
<td>Q_FIRST_CELL</td>
<td>25C47430</td>
</tr>
<tr>
<td>POOL_NUM_GET_TOTAL</td>
<td>000000F0</td>
</tr>
<tr>
<td>POOL_NUM_FREE</td>
<td>000000003</td>
</tr>
<tr>
<td>POOL_INDEX_SAME_SIZE</td>
<td>000005</td>
</tr>
<tr>
<td>POOL_INDEX_SIZE</td>
<td>000005</td>
</tr>
<tr>
<td>POOL_TRACE_TABLE</td>
<td>25DA6140</td>
</tr>
</tbody>
</table>

**Heap Pool Extent Mapping**

**EXTENT:** 25C1F500

To display entire pool extent: IP LIST 25C1F500 LEN(X'00001028') ASID(X'0020')

- **25C47430:** Free storage cell. To display: IP LIST 25C47430 LEN(X'00000408') ASID(X'0020')
- **25C47838:** Free storage cell. To display: IP LIST 25C47838 LEN(X'00000408') ASID(X'0020')
- **25C47C70:** Free storage cell. To display: IP LIST 25C47C70 LEN(X'00000408') ASID(X'0020')

No errors were found while processing free chain.

Summary of analysis for Pool 5.3:

- Number of cells: 1 Free: 3 Allocated: 0 Total Used: 4
- 00000000 free cells were not accounted for.
- No errors were found while processing this Pool.

### Data for pool 5.4:

**Data for pool 5.4:**

**POOLDATA:** 25C1F600

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOL_INDEX</td>
<td>00000009</td>
</tr>
<tr>
<td>INPUT_CELL_SIZE</td>
<td>000000400</td>
</tr>
<tr>
<td>CELL_INDEX</td>
<td>00000000</td>
</tr>
<tr>
<td>CELL_POOL_SIZE</td>
<td>00001020</td>
</tr>
<tr>
<td>NEXT_CELL</td>
<td>25C48048</td>
</tr>
<tr>
<td>POOL_EXTENSION</td>
<td>00000001</td>
</tr>
<tr>
<td>INPUT_PERCENT</td>
<td>00000001</td>
</tr>
<tr>
<td>CELL_POOL_INDEX</td>
<td>00000000</td>
</tr>
<tr>
<td>CELL_POOL_NUM</td>
<td>00000000004</td>
</tr>
<tr>
<td>LAST_CELL</td>
<td>25C49078</td>
</tr>
<tr>
<td>POOL_LATCH_ADDR</td>
<td>25C54C24</td>
</tr>
<tr>
<td>Q_CONTROL_INFO</td>
<td>0000001D</td>
</tr>
<tr>
<td>Q_FIRST_CELL</td>
<td>25C48460</td>
</tr>
<tr>
<td>POOL_NUM_GET_TOTAL</td>
<td>000000F0</td>
</tr>
<tr>
<td>POOL_NUM_FREE</td>
<td>000000003</td>
</tr>
<tr>
<td>POOL_INDEX_SAME_SIZE</td>
<td>000005</td>
</tr>
<tr>
<td>POOL_INDEX_SIZE</td>
<td>000005</td>
</tr>
<tr>
<td>POOL_TRACE_TABLE</td>
<td>25DD6160</td>
</tr>
</tbody>
</table>

**Heap Pool Extent Mapping**

**EXTENT:** 25C1F600

To display entire pool extent: IP LIST 25C1F600 LEN(X'00001028') ASID(X'0020')

- **25C48460:** Free storage cell. To display: IP LIST 25C48460 LEN(X'00000408') ASID(X'0020')
- **25C48868:** Free storage cell. To display: IP LIST 25C48868 LEN(X'00000408') ASID(X'0020')
- **25C48C70:** Free storage cell. To display: IP LIST 25C48C70 LEN(X'00000408') ASID(X'0020')

No errors were found while processing free chain.

Summary of analysis for Pool 5.4:

- Number of cells: 1 Free: 3 Allocated: 0 Total Used: 4
- 00000000 free cells were not accounted for.
- No errors were found while processing this Pool.

### Data for pool 5.5:

**Data for pool 5.5:**

**POOLDATA:** 25C1F600

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOL_INDEX</td>
<td>00000009</td>
</tr>
<tr>
<td>INPUT_CELL_SIZE</td>
<td>000000400</td>
</tr>
<tr>
<td>CELL_INDEX</td>
<td>00000000</td>
</tr>
<tr>
<td>CELL_POOL_SIZE</td>
<td>00001020</td>
</tr>
<tr>
<td>NEXT_CELL</td>
<td>25C49078</td>
</tr>
<tr>
<td>POOL_EXTENSION</td>
<td>00000001</td>
</tr>
<tr>
<td>INPUT_PERCENT</td>
<td>00000001</td>
</tr>
<tr>
<td>CELL_POOL_INDEX</td>
<td>00000000</td>
</tr>
<tr>
<td>CELL_POOL_NUM</td>
<td>00000000004</td>
</tr>
<tr>
<td>LAST_CELL</td>
<td>25C49870</td>
</tr>
<tr>
<td>POOL_LATCH_ADDR</td>
<td>25C54C24</td>
</tr>
<tr>
<td>Q_CONTROL_INFO</td>
<td>0000001D</td>
</tr>
<tr>
<td>Q_FIRST_CELL</td>
<td>25C494960</td>
</tr>
<tr>
<td>POOL_NUM_GET_TOTAL</td>
<td>000000F0</td>
</tr>
<tr>
<td>POOL_NUM_FREE</td>
<td>000000003</td>
</tr>
<tr>
<td>POOL_INDEX_SAME_SIZE</td>
<td>000005</td>
</tr>
<tr>
<td>POOL_INDEX_SIZE</td>
<td>000005</td>
</tr>
<tr>
<td>POOL_TRACE_TABLE</td>
<td>25DD6160</td>
</tr>
</tbody>
</table>

**Heap Pool Extent Mapping**

**EXTENT:** 25C1F600

To display entire pool extent: IP LIST 25C1F600 LEN(X'00001028') ASID(X'0020')

- **25C494960:** Free storage cell. To display: IP LIST 25C494960 LEN(X'00000408') ASID(X'0020')
- **25C49870:** Free storage cell. To display: IP LIST 25C49870 LEN(X'00000408') ASID(X'0020')

No errors were found while processing free chain.

Summary of analysis for Pool 5.5:

- Number of cells: 1 Free: 3 Allocated: 0 Total Used: 4
- 00000000 free cells were not accounted for.
- No errors were found while processing this Pool.

### Data for pool 6:

**Data for pool 6:**

**POOLDATA:** 25C1F700

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOL_INDEX</td>
<td>00000006A</td>
</tr>
<tr>
<td>INPUT_CELL_SIZE</td>
<td>000000008</td>
</tr>
<tr>
<td>CELL_INDEX</td>
<td>00000000</td>
</tr>
<tr>
<td>CELL_POOL_SIZE</td>
<td>00008020</td>
</tr>
<tr>
<td>NEXT_CELL</td>
<td>25C49870</td>
</tr>
<tr>
<td>POOL_EXTENSION</td>
<td>00000001</td>
</tr>
<tr>
<td>INPUT_PERCENT</td>
<td>00000001</td>
</tr>
<tr>
<td>CELL_POOL_INDEX</td>
<td>00000000</td>
</tr>
<tr>
<td>CELL_POOL_NUM</td>
<td>00000000004</td>
</tr>
</tbody>
</table>

**Heap Pool Extent Mapping**

**EXTENT:** 25C1F700

To display entire pool extent: IP LIST 25C1F700 LEN(X'00001028') ASID(X'0020')

- **25C49870:** Free storage cell. To display: IP LIST 25C49870 LEN(X'00000408') ASID(X'0020')

No errors were found while processing free chain.

Summary of analysis for Pool 6:

- Number of cells: 1 Free: 3 Allocated: 0 Total Used: 4
- 00000000 free cells were not accounted for.
- No errors were found while processing this Pool.
Heap pools report sections of the LEDATA output

The heap pools report provides information about the following items:

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

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

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Free Chain Validation</td>
<td>Within each cell pool, Language Environment keeps track of unallocated cells by chaining them together. The LEDATA HEAP option validates the free chain within each cell pool. It verifies that the cell pointer is within a valid extent and that the cell pool number is valid. If the formatter finds a problem, it will place an error message describing the problem directly after the formatted line of the cell that failed validation.</td>
</tr>
<tr>
<td>[2] Heap Pool Extent Mapping Report</td>
<td>The LEDATA HEAP option produces a report that lists all of the cells within each pool extent, and identifies the cells as either allocated or freed. For each allocated cell, the contents of the first X'20' bytes of the area are displayed to identify the reason for the storage allocation. The formatter validates if cell pool number in header is correct.</td>
</tr>
</tbody>
</table>

Understanding the heap pools trace LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap pools trace report when the HPT option is used. The argument **value** is the ID of the pool to be formatted in the report. Table 24 on page 117 describes the contents of the report.
Table 24: Contents of heap pools trace section of LEDATA output

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2] Pool Information</td>
<td>Information includes the number of the pool (POOLID) that is currently being formatted, the ASID, and the number of entries formatted and the total number of entries taken. <strong>Note:</strong> The trace wraps for each poolid after a specific number of entries. The number of entries is controlled by the HEAPCHK runtime option.</td>
</tr>
<tr>
<td>[3] Timestamp</td>
<td>The time this trace entry was taken. The trace entries are formatted in reverse order (most recent trace entry first).</td>
</tr>
</tbody>
</table>
Table 24: Contents of heap pools trace section of LEDATA output (continued)

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4] Trace Table Entry</td>
<td>contents</td>
</tr>
<tr>
<td></td>
<td>The individual trace entry:</td>
</tr>
<tr>
<td></td>
<td>• The TYPE - GET or FREE.</td>
</tr>
<tr>
<td></td>
<td>• The Cell within the pool being acted upon.</td>
</tr>
<tr>
<td></td>
<td>• The CPU and TCB which requested or freed the cell.</td>
</tr>
<tr>
<td></td>
<td>• A traceback at the time of the request. The number of entries in this traceback is limited by the HEAPCHK runtime option.</td>
</tr>
</tbody>
</table>

Understanding the C/C++-specific LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of C/C++-specific control blocks from a system dump when the COMP(C), COMP(ALL) , or ALL parameter is specified and C/C++ is active in the dump. The following example illustrates the C/C++-specific output produced. Figure 8 on page 41 and Table 25 on page 136 describe the information contained in the formatted output. Ellipses are used to summarize some sections of the dump. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.

************ CRTL ENVIRONMENT DATA ************

[C] CENG: 20C13A10
+0000C7 OS_SCTYPE:00000000 CGENE:20C10B90 CRENRT:20C14FF8
+0001C8 CFLTINT:4E000000 00000000 CPRMS:20C127A8 TRAEC:00000000
+0002D8 CTHD:20C08E8A CURR_FECB:20C101D8 CEDCXV:A0DF6500
+0002DA CGEN_CPCB:20C0DE8A CGEN_CEDB:20C0F938 CFLG3:00
+0002E0 CIO:20C0E098 FDSETFD:00000000 FCB_MUTEXOK:0000
+0006CC T_C16:00000000 T_C17:00000000 CEDCV:2117A9A8
+000288 CTOFSV:00000000 TRTSPACE:20C0EE50

[C] CGENE: 20C10B90
+000000 CENYEE:GENE GENESIZE:000006E0 CGENEPTR:20C10B90
+0000D0 CERRNO:00000000 TEMPLONG:00000000 AMRC:20C0E770
+000104 STDINFILE:20C0F718 STDOUTFILE:20C0F2D8
+00010C STDERRFILE:20C0F4F8 CTYPE:21035412 LC_CTYPE:21035412
+000124 LC_CHARMAP:21035F00 MIN_FLT:00100000 00000000 00000000 00000000
+000510 MAX_FLT:7FFFFFFF FFFFFFFF 71FFFFFF FFFFFFFF
+000528 FLT_EPS:3C100000 00000000 DBL_EPS:34100000 00000000
+000544 LDBL_EPS:26100000 00000000 18000000 00000000
+000554 IMPCBLIST:20C145A4 ADDRTBL:20C0E2C8
+0006D4 ABND_CODE:00000000 RSN_CODE:00000000

[C] CEBD: 20C0F938
+000000 EYE:CEBD SIZE:000007C0 PTR:20C0F938 CLLST:20C00568
+000010 EELANG:0003 CASWITCH:0000 CLWA:20C11270
+000018 CALTWI:20C115C0 CCADDR:20C00A10 CFLGS:00000000
+000024 CEESTART:20C00000 CANCHOR:00000000 RPLEN:00000000
+000030 ACBLEN:00000000 LC:20C10100 VALID_HIGH:210306C0
+00003C _LOW:2102FE8 HEAD_FECB:00000000 ATEXIT_COUNT:00000000
+000048 _EMPTY_COUNT:20C0FA10 MAINPRMS:2132D768
+000058 STDDNF:20C0F718 STOUTFILE:20C0F2D8
+000064 CINFO:20C10190 CMS_WRITE_DISK:4040 DISK_SET:00000000
+000070 MIN_FLT:00100000 00000000 00000000 00000000
+000080 MAX_FLT:7FFFFFFF 71FFFFFF FFFFFFF
+000090 FLT_EPS:3C100000 00000000 00000000
+0000A0 LDBL_EPS:26100000 00000000 00000000 00000000
+0000B0 MTF_MAINTASK_BLK:00000000 EMSS_SETTING:00 DEPTH:00000000
+0000CC SCREEN_WIDTH:00000000 USERID:CHARUM.
+0000CC HEAP24_ANCHOR:00000000 TCIC:00000000 TKCLI:00000000
+0000D8 ATEXIT_FUNCS01:20C0FA24 00000000 00000000 00000000 00000000
+0000EC ATEXIT_FUNCS02:20C0FA38 00000000 00000000 00000000 00000000 00000000
+000100 ATEXIT_FUNCS03:20C0FA4C 00000000 00000000 00000000 00000000 00000000
+000114 ATEXIT_FUNCS04:20C0FA60 00000000 00000000 00000000 00000000 00000000
+000128 ATEXIT_FUNCS05:20C0FA74 00000000 00000000 00000000 00000000 00000000
+00013C ATEXIT_FUNCS06:20C0FA88 00000000 00000000 00000000 00000000 00000000
+000150 ATEXIT_FUNCS07:20C0FA9C 00000000 00000000 00000000 00000000 00000000
+000164 ATEXIT_FUNCS08:20C0FA80 00000000 00000000 00000000 00000000 00000000
+000178 ATEXIT_FUNCS09:20C0FA4C 00000000 00000000 00000000 00000000 00000000

118 z/OS: Language Environment Debugging Guide
Exiting CRTL Environment Data

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

CIO: 20C0E098
+000000 EYE:CIO SIZE:00000090 PTR:00000000 FLG1:09
+00000D FLG2:E0 FLG3:00 FLG4:00 DUMMYF:20C0E128
+000014 EDC2Z:00000000 FCBSTART:2135A040 DUMMYFCB:20C0E148
+000020 IOANYLIST:2135A000
+000028 IOBELOWLIST:00014000 FCBDDLIST:00000000
+000030 PERRORBUF:20C0DF60 TMPCOUNTER:00000000
+000038 TEMPMEM:00000000 PROMPTBUF:00000000 IO24:000122D0
+000044 IOEXITS:000127C8 TERMINALCHAIN:00000000
+00004C VANCHOR:00000000 XTI:00000000 ENOWP24:20F484A0
+000058 MAXNUMDESCRPS:00000000 DESCARRAY:00000000
+000064 TEMPFILENUM:00000000 CSS:00000000 DUMMY_NAME:........
+000074 HOSTNAME_CACHE:00000000 HOSTADDR_CACHE:00000000
+00007C IO31:20E62508
+000080 LAST_FD_CLOSE:00000000 00000000 00000000 00000000
+000090 IOGET64:213244E8 IOFREE64:21323D80

FFIL: 2135A020
+000000 MARKER2:AFCB FILE:00000000 __FP:2135A040
+00000C MARKER2:AFCBFCB FF_FLAGS:01000000
+000014 FCBMUTEX:00000000 THREADID:00000000 00000000
+000028 IOBELOWLIST:00000000 DUMMYFCB:20C0E148
+000038 TEMPMEM:00000000 PROMPTBUF:00000000 IO24:000122D0
+000048 IOEXITS:00000000 TERMINALCHAIN:00000000
+000050 MAXNUMDESCRPS:00000000 DESCARRAY:00000000
+000060 TEMPFILENUM:00000000 CSS:00000000 DUMMY_NAME:........
+000070 HOSTNAME_CACHE:00000000 HOSTADDR_CACHE:00000000
+000082 IO31:20E62508
+000090 LAST_FD_CLOSE:00000000 00000000 00000000 00000000
+000098 IOGET64:213244E8 IOFREE64:21323D80

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

Using Language Environment debugging facilities 121
File name: CHARUM.A.B

File name: CHARUM.B.C

File name: CHARUM.A.B

File name: CHARUM.B.C
Using Language Environment debugging facilities 129
Using Language Environment debugging facilities 131
Exiting CRTL I/O Control Blocks
Exiting Language Environment Data

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

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

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7] File Control Blocks</td>
<td>Formats the C/C++ file control block (FCB). The FCB and its related control blocks represent the information needed by each open stream. The following related control blocks are included.</td>
</tr>
<tr>
<td></td>
<td><strong>FFIL</strong> Formats the header of the C/C++ file control block (FCB).</td>
</tr>
<tr>
<td></td>
<td><strong>FSCE</strong> The file specific category extension control block (FSCE), which represents the specific type of IO being performed. The following is a list of FSCEs that may be formatted; other FSCEs will be displayed using a generic overlay.</td>
</tr>
<tr>
<td></td>
<td><strong>HFSF</strong> UNIX file system file</td>
</tr>
<tr>
<td></td>
<td><strong>HSPF</strong> Hiper-Space file</td>
</tr>
<tr>
<td></td>
<td><strong>INTC</strong> Intercept file</td>
</tr>
<tr>
<td></td>
<td><strong>MEMF</strong> Memory file</td>
</tr>
<tr>
<td></td>
<td><strong>OSNS</strong> OS no seek</td>
</tr>
<tr>
<td></td>
<td><strong>OSFS</strong> OS fixed text</td>
</tr>
<tr>
<td></td>
<td><strong>OSVF</strong> OS variable text</td>
</tr>
<tr>
<td></td>
<td><strong>OSUT</strong> OS undefined format text</td>
</tr>
<tr>
<td></td>
<td><strong>TDQF</strong> CICS Transient Data Queue file</td>
</tr>
<tr>
<td></td>
<td><strong>TERM</strong> Terminal file</td>
</tr>
<tr>
<td></td>
<td><strong>VSAM</strong> VSAM file</td>
</tr>
<tr>
<td></td>
<td><strong>OSIO</strong> The OS IO interface control block.</td>
</tr>
<tr>
<td></td>
<td><strong>OSIOE</strong> The OS IO extended interface control block.</td>
</tr>
<tr>
<td></td>
<td><strong>DCB</strong> The data control block. For more information about the DCB, see <a href="https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary">z/OS DFSMS Macro Instructions for Data Sets</a>.</td>
</tr>
<tr>
<td></td>
<td><strong>DCBE</strong> The data control block extension. For more information about the DCBE, see <a href="https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary">z/OS DFSMS Macro Instructions for Data Sets</a>.</td>
</tr>
<tr>
<td></td>
<td><strong>JFCB</strong> The job file control block (JFCB). For more information about the JFCB, see <a href="https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary">z/OS MVS Data Areas in the z/OS Internet library</a>.</td>
</tr>
<tr>
<td></td>
<td><strong>JFCBX</strong> The job file control block extension (JFCBX).</td>
</tr>
<tr>
<td></td>
<td><strong>MBUF</strong> The message buffer control block (MBUF).</td>
</tr>
<tr>
<td>[8] Memory File Control Blocks</td>
<td>This section formats the C/C++ memory file control block (MFCB).</td>
</tr>
</tbody>
</table>
Understanding the COBOL-specific LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of COBOL-specific control blocks from a system dump when the COMP(COBOL), COMP(ALL) or ALL parameter is specified and COBOL is active in the dump. The following example illustrates the COBOL-specific output produced.

The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option. Table 26 on page 139 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.

**************************************************************************
COBOL ENVIRONMENT DATA
**************************************************************************

[1] RUNCOM: 00049038
+000000 IDENT:C3RUNCOM LENGTH:000002D8 FLAGS:00860000
+000018 RU_ID:000178B0 INVK_RSA:00005F80
+000024 MAIN_PGM_ADDR:00007DE8 PARM_ADDR:000179D0
+00003C ITBNAB:00000000 DUM_CLLE:0BF15BA8
+000040 THDCOM:0001AA80 COBVEC:0001A1BC
+00004C COBVEC2:0001A7FC CAA:00018920
+000058 UPSI_SWITCHES:00000000
+000064 LEDATA:0BF159A8
+000070 MAIN_PGM_CLLE:00049328
+000078 NEXT_RUNCOM:00000000
+000080 THDCOM:0001AA80
+00008C COBVEC:0001A1BC
+000094 COBVEC2:0001A7FC SUBCOM:00000000
+0000A0 ITBLK_TRAP_RSA:00000000
+0000A4 ITBLK_NAB:00000000
+0000A8 DUM_MAIN_DSA:00000000
+0000B4 RRE_TAIL_RSA:00000000
+0000C0 TGT_ADDR:00050248
+000000 IDENT:C3THDCOM LENGTH:000001E8 FLAGS:8100000000000000
+000018 COBCOM:0001A108 1ST_RUNCOM:00049038
+000028 1ST_PROGRAM:CALLSUBX
+000034 CEEINT_PLIST:00000000 00000000 00000000 00000000 00000000
+00004C ---------->:00000000 00000000 00000000 00000000 00000000
+000058 THDCOM:0001AA80 INSH:00000000
+000070 LRR_THDCOM:00000000
+000078 LRR_ITBLK:00000000
+000000 IDENT:C3COBCOM LENGTH:00000978 VERSION:010900
+000058 FLAGS:90600000 COBVEC:0001A1BC
+000064 LOADPG:00000100 CICS_EIB:00000000
+000070 THDCOM:0001AA80 CALC_RSA:00000000
+000078 THDCOM:0001A80 LRR_CBCOM:00000000
+000084 ITBLK:00000000
+00008C CICS_EIB:00000000
+000094 SORT_RETURN:00000000
+000000 LENGNAME:PARM5 OPEN_NON_EXT_FILES:0000
+00000C LANG_LST:00050F98 INFO_FLAGS:8891
+000018 TGT_ADDR:00050248 LE_TOKEN:0BF150BC
+000000 IDENT:3TG LVL:05
+00000C LANG_LST:00050F98 INFO_FLAGS:8891
+000018 TGT_ADDR:00050248 LE_TOKEN:0BF150BC

138 z/OS: Language Environment Debugging Guide
Table 26: Contents of COBOL-specific sections of LEDATA Output

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

Exiting COBOL Environment Data
The names of the control blocks have changed in Enterprise COBOL V5.1. Table 27 on page 141 shows the correspondence with COBOL V4R2 and prior releases.

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

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of PL/I-specific control blocks from a system dump when the COMP(PLI), COMP(ALL) or ALL parameter is specified and PL/I is active in the dump. The following example illustrates the PL/I-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option. Table 28 on page 147 describes the information contained in the formatted output. For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows.

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

[1] RXRCB: 00021000
   +000000  ID:ZRCB      LIBVEC:00013038  RSAP:00000000
   +000010  PSMA:00000000  PSLA:00000000

   +000000  ID:ZPRB      RCB:00021000  SYSP_FCB:00063004
   +000010  MSGF_FCB:00000000  PRV_INIT:00016B38
   +00001C  ENT_FCO:00000000  MSGFFLAGS:00  FLAGS:B6000000
   +000030  DCL_LIST:00000000  DBG_STG:00000000
   +000038  DCL_LIST_LEN:00000000

[3] TIA: 00056298
   +000000  TISA:00000000  TAPC:00000000  TERA:00000000  TINM:0000
   +00000E  TFL1:0030  TWTW:00000000  TEXF:00000000  TLFE:00000000
   +000020  TDUB:00056450  TDDS:00000000  TXRAD:00000000
   +00002C  TASM:00000000  TSNM:........  TASR:00000000
   +00003C  TFEP:000571C8  TAST:00000000  TERC:00000000
   +000048  TCTF:00000000  TCTL:00000000  ATAC:00033B88
   +000054  TABD:00000000  TRPS:00000000  TFL3:00000000  TCPM:00
   +000060  TXRES:00000000  TXIIC:00000000  TXHIN:00000000
   +00006C  TXHIC:00000000  TXBOC:00000000  TXLFE:00000000
   +000084  RC:00000000  REASON:00000000  ADBG:00000000
   +000090  PARM:00000000  PADDR:00000000  PLIST:00000000
   +00009C  STRLOC:00000000  STRLEN:0000  STRVAR:0000
   +0000A4  ZEROSTR:00000000  PRECALL:00000000
   +0000B0  PRETERM:00000000  FCB:00000000  PREINT:151016F0
   +0000BD  USERWD:00000000  CCPARM:00000000  MAINLAN:0000
   +0000C6  MSG_LECL:00000000  MSG_RT:00000000  TPMV:00000000
   +0000D0  TMSK:00000000  USERCODE:00000000  PREREINT:151016F0
   +0000DC  SYSP_DCL:00000000  PG_ADDR:00000000  PG_ADDR:00000000
   +0000EC  PG_PLIST:00000000  PNUM:7FFFFFFF

[4] FECB: 000571C8
   +000000  CHAIN:00000000  PRVO:0A000000  NAME:IEFBR14
   +000010  CODE:50E0D068 58FF0014 41C0C000 0CEF58E0 D0680B0E
   +000024  EPA:00E73000  SAVE:00000000  END:00000008
   +000030  FLAGS:08000000  MENTRY:00000000  LOADPTR:00000000

   +000000  FCB:00000000  ATT:02010400  OPA:01861800
   +00000C  ENV:15101770  FNLEN:0007  FNLEN:0007
   +000010  CODE:500E0006 58FF0014 41C00000 0CEF58E0 D068008E
   +000024  EPA:00E73000  SAVE:00000000  END:00000008
   +000030  FLAGS:08000000  MENTRY:00000000  LOADPTR:00000000

FILENAME:DDVSAM1

**Additional data for file 1**

FILENAME:DDVSAM1

ATTRS:RECORD OUTPUT SEQ KEYED BUF
Using Language Environment debugging facilities
Using Language Environment debugging facilities 145
Table 28: Contents of PL/I-specific sections of LEDATA output

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

Formatting individual control blocks

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

address

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

The name of the control block to be formatted. The control blocks that can be individually formatted are listed in Table 29 on page 148. In general, the name of each control block is similar to that used by the LEDATA verb exit and is generally found in the control block's eyecatcher field. However, all control block names are prefixed with "CEE" to uniquely define the Language Environment control block names to IPCS.

For example, the following command produces the output shown in Figure 34 on page 148.

```plaintext
CBF 213F6B48 struct(CEECAA)
```

Figure 34: 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.

### Table 29: Language Environment Control blocks that can be individually formatted

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEADHP</td>
<td>Additional Heap Control Block</td>
</tr>
<tr>
<td>CEECAA</td>
<td>Common Anchor Area</td>
</tr>
<tr>
<td>CEECIB</td>
<td>Condition Information Block</td>
</tr>
<tr>
<td>Control Block</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</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>
<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>CEEMDIST</td>
<td>Message Destination</td>
</tr>
<tr>
<td>CEEMGF</td>
<td>Mapping of the Message Formatter (IBM1MGF)</td>
</tr>
<tr>
<td>CEEPCB</td>
<td>Process Control Block</td>
</tr>
<tr>
<td>CEEPIMCB</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>

**Controlling access to CEEDUMPs and DYNDUMPs**

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

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

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

- A user is running a Language Environment application as a RACF-controlled program on a system where the IEAABD.DMPAUTH resource has been defined, but the user has not been permitted access to this resource.
• A user is running an authorized key Language Environment application in a non-started task address space but the user has not been permitted access to the IEAABD.DMPAKEY resource.

• A user is running a Language Environment application in a non-started task address space that has the JSCBPASS indicator on, including applications whose PPT entry specifies bypassing security protection.

When Language Environment has suppressed a dump, message CEE3880I will be written to the application’s programmer log. To allow the user to receive this dump, the user may need to be permitted to the IEAABD.DMPAUTH or IEAABD.DMPAKEY resource. For more information, refer to the documentation for message CEE3880I in the z/OS Language Environment Runtime Messages, as well as the z/OS Security Server RACF Security Administrator’s Guide.

### Requesting a Language Environment trace for debugging

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

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

For more information about the CEE3DMP callable service, the TERMTHDACT runtime option, or the TRACE runtime option, see z/OS Language Environment Programming Reference.

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

The contents of the Language Environment dump depend on the values set in the TERMTHDACT runtime option. Table 30 on page 150 summarizes the dump contents that are generated under abnormal termination.

<table>
<thead>
<tr>
<th>TERMTHDACT value</th>
<th>Type of dump generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMTHDACT(QUIET)</td>
<td>Language Environment dump containing the trace table only</td>
</tr>
<tr>
<td>TERMTHDACT(MSG)</td>
<td>Language Environment dump containing the trace table only</td>
</tr>
<tr>
<td>TERMTHDACT(TRACE)</td>
<td>Language Environment dump containing the trace table and the traceback</td>
</tr>
<tr>
<td>TERMTHDACT(DUMP)</td>
<td>Language Environment dump containing thread/enclave/process storage and control blocks (the trace table is included as an enclave control block)</td>
</tr>
<tr>
<td>TERMTHDACT(UAONLY)</td>
<td>System dump of the user address space and a Language Environment dump that contains the trace table</td>
</tr>
<tr>
<td>TERMTHDACT(UATRACE)</td>
<td>Language Environment dump that contains traceback information, and a system dump of the user address space</td>
</tr>
<tr>
<td>TERMTHDACT(UADUMP)</td>
<td>Language Environment dump containing thread/enclave/process storage and control blocks (the trace table is included as an enclave control block), and a user address space dump</td>
</tr>
</tbody>
</table>
Table 30: TERMTHDACT runtime option settings and dump contents produced (continued)

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

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

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

### Locating the trace dump

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

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

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

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

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

```
/path/Fname.Date.Time.Pid
```

- **path**
  - Path determined from the above algorithm.

- **Fname**
  - Name specified in the FNAME parameter on the call to CEE3DMP (default is CEEDUMP).

- **Date**
  - Date the dump is taken, appearing in the format YYYYMMDD (such as 20090307 for March 7, 2009).

- **Time**
  - Time the dump is taken, appearing in the format HHMMSS (such as 175501 for 05:55:01 p.m.).

- **Pid**
  - Process ID the application is running in when the dump is taken.
Using the Language Environment trace table format in a dump report

The Language Environment trace table is established unconditionally at enclave initialization time if the TRACE runtime option is set to ON. All threads in the enclave share the trace table; there is no thread-specific table, nor can the table be dynamically extended or enlarged.

Understanding the trace table entry (TTE)

Each trace table entry is a fixed-length record consisting of a fixed-format portion (containing such items as the timestamp, thread ID, and member ID) and a member-specific portion. The member-specific portion has a fixed length, of which some (or all) can be unused. For information about how participating products use the trace table entry, see the product-specific documentation. The format of the trace table entry is as follows:

<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 35: Format of the trace table entry

Time
The 64-bit value obtained from a store clock (STCK).

Thread ID
The 8-byte thread ID of the thread that is adding the trace table entry.

Member ID and Flags
Contains 2 fields:

Member ID
The 1-byte member ID of the member making the trace table entry, as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>CEL</td>
</tr>
<tr>
<td>03</td>
<td>C/C++</td>
</tr>
<tr>
<td>04</td>
<td>COBOL V5 (and later releases)</td>
</tr>
<tr>
<td>05</td>
<td>COBOL</td>
</tr>
<tr>
<td>07</td>
<td>Fortran</td>
</tr>
<tr>
<td>08</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>PL/I</td>
</tr>
<tr>
<td>11</td>
<td>Enterprise PL/I</td>
</tr>
<tr>
<td>12</td>
<td>Sockets</td>
</tr>
</tbody>
</table>

Flags
24 flags reserved for internal use.
**Member Entry Type**

A number that indicates the type of the member-specific trace information that follows the field. To uniquely identify the information contained in a specific TTE, you must consider Member ID as well as Member Entry Type.

**Member-Specific Information**

Based on the member ID and the member entry type, this field contains the specific information for the entry, up to 104 bytes. For C/C++, the entry type of 1 is a record that records an invocation of a base C runtime library function. The entry consists of the name of the invoking function and the name of the invoked function. Entry type 2 is a record that records the return from the base library function. It contains the returned value and the value of errno.

**Member-specific information in the trace table entry**

Global tracing is activated by using the LE=n suboption of the TRACE runtime option. This requests all Language Environment members to generate trace records in the trace table. The settings for the global trace events are:

- **Level**
  - **Description**
  - **0**
    - No global trace
  - **1**
    - Trace all runtime library (RTL) function entry and exits
  - **2**
    - Trace all RTL mutex init/destroy and lock/unlock
  - **3**
    - Trace all RTL function entry and exits, and all mutex init/destroy and lock/unlock
  - **8**
    - Trace all RTL storage allocation/deallocation
  - **20**
    - Trace all XPLINK/non-XPLINK transitions for AMODE 31 only. If `#pragma linkage (xxxxxxxx, OS_UPSTACK)` is specified, no transitions are recorded.

**When LE=1 is specified**

Table 31 on page 153 shows the C/C++ records that may be generated. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 195.

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

**When LE=2 is specified**

Table 32 on page 153 shows the Language Environment records that 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>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>00000102</td>
<td>LT</td>
<td>R</td>
<td>Latch Release</td>
</tr>
<tr>
<td>01</td>
<td>00000103</td>
<td>LT</td>
<td>W</td>
<td>Latch Wait</td>
</tr>
<tr>
<td>01</td>
<td>00000104</td>
<td>LT</td>
<td>AW</td>
<td>Latch Acquire after Wait</td>
</tr>
<tr>
<td>01</td>
<td>00000106</td>
<td>LT</td>
<td>I</td>
<td>Latch Increment (Recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000107</td>
<td>LT</td>
<td>D</td>
<td>Latch Decrement (Recursive)</td>
</tr>
<tr>
<td>01</td>
<td>000002FC</td>
<td>LE</td>
<td>EUO</td>
<td>Latch unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000002FD</td>
<td>LE</td>
<td>EO</td>
<td>Latch already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>00000301</td>
<td>MX</td>
<td>A</td>
<td>Mutex acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000302</td>
<td>MX</td>
<td>R</td>
<td>Mutex release</td>
</tr>
<tr>
<td>01</td>
<td>00000303</td>
<td>MX</td>
<td>W</td>
<td>Mutex wait</td>
</tr>
<tr>
<td>01</td>
<td>00000304</td>
<td>MX</td>
<td>AW</td>
<td>Mutex acquire after wait</td>
</tr>
<tr>
<td>01</td>
<td>00000305</td>
<td>MX</td>
<td>B</td>
<td>Mutex busy (Trylock failed)</td>
</tr>
<tr>
<td>01</td>
<td>00000306</td>
<td>MX</td>
<td>I</td>
<td>Mutex increment (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000307</td>
<td>MX</td>
<td>D</td>
<td>Mutex decrement (recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000315</td>
<td>MX</td>
<td>IN</td>
<td>Mutex initialize</td>
</tr>
<tr>
<td>01</td>
<td>00000316</td>
<td>MX</td>
<td>DS</td>
<td>Mutex destroy</td>
</tr>
<tr>
<td>01</td>
<td>0000031D</td>
<td>MX</td>
<td>BI</td>
<td>Shared memory lock init</td>
</tr>
<tr>
<td>01</td>
<td>0000031E</td>
<td>MX</td>
<td>BD</td>
<td>Shared memory lock destroy</td>
</tr>
<tr>
<td>01</td>
<td>0000031F</td>
<td>MX</td>
<td>BO</td>
<td>shared memory lock obtain</td>
</tr>
<tr>
<td>01</td>
<td>00000320</td>
<td>MX</td>
<td>BC</td>
<td>Shared memory lock obtain on condition</td>
</tr>
<tr>
<td>01</td>
<td>00000321</td>
<td>MX</td>
<td>BR</td>
<td>Shared memory lock release</td>
</tr>
<tr>
<td>01</td>
<td>00000324</td>
<td>MX</td>
<td>CIN</td>
<td>Call to SMC_INIT</td>
</tr>
<tr>
<td>01</td>
<td>00000325</td>
<td>MX</td>
<td>CSD</td>
<td>Call to SMC_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>00000326</td>
<td>MX</td>
<td>CSO</td>
<td>Shared resource obtain</td>
</tr>
<tr>
<td>01</td>
<td>00000327</td>
<td>MX</td>
<td>CSR</td>
<td>Shared resource release</td>
</tr>
<tr>
<td>01</td>
<td>00000328</td>
<td>MX</td>
<td>CST</td>
<td>Call to SMC.SetupToWait</td>
</tr>
<tr>
<td>01</td>
<td>00000329</td>
<td>MX</td>
<td>CSP</td>
<td>Call to SMC_POST</td>
</tr>
<tr>
<td>01</td>
<td>000004CC</td>
<td>ME</td>
<td>FFR</td>
<td>Error - Forced release (shared mutex)</td>
</tr>
<tr>
<td>01</td>
<td>000004CD</td>
<td>ME</td>
<td>FFD</td>
<td>Error - Forced decrement (shared mutex)</td>
</tr>
<tr>
<td>01</td>
<td>000004CE</td>
<td>ME</td>
<td>FBD</td>
<td>Error - BPX_SMC(DESTROY) error return</td>
</tr>
<tr>
<td>01</td>
<td>000004CF</td>
<td>ME</td>
<td>FBU</td>
<td>Error - BPX_SMC(fail) returns EBUSY</td>
</tr>
<tr>
<td>01</td>
<td>000004D0</td>
<td>ME</td>
<td>FIV</td>
<td>Error - BPX_SMC(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000004D4</td>
<td>ME</td>
<td>FDU</td>
<td>Error - Destroy failed (uninitialized) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000004D5</td>
<td>ME</td>
<td>FP</td>
<td>Error - Program check (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000004DB</td>
<td>ME</td>
<td>ESC</td>
<td>Error - BPX1SMC error return</td>
</tr>
<tr>
<td>01</td>
<td>000004DE</td>
<td>ME</td>
<td>EDL</td>
<td>Shared memory lock returns deadlock</td>
</tr>
<tr>
<td>01</td>
<td>000004DF</td>
<td>ME</td>
<td>EIV</td>
<td>Shared memory lock returns invalid</td>
</tr>
<tr>
<td>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>000004E0</td>
<td>ME</td>
<td>EPM</td>
<td>Shared memory lock returns eperm</td>
</tr>
<tr>
<td>01</td>
<td>000004E1</td>
<td>ME</td>
<td>EAG</td>
<td>Shared memory lock returns eagain</td>
</tr>
<tr>
<td>01</td>
<td>000004E2</td>
<td>ME</td>
<td>EBU</td>
<td>Shared memory lock returns ebusy</td>
</tr>
<tr>
<td>01</td>
<td>000004E3</td>
<td>ME</td>
<td>ENM</td>
<td>Shared memory lock returns enomem</td>
</tr>
<tr>
<td>01</td>
<td>000004E4</td>
<td>ME</td>
<td>EBR</td>
<td>Shared memory lock release error</td>
</tr>
<tr>
<td>01</td>
<td>000004E5</td>
<td>ME</td>
<td>EBC</td>
<td>Shared memory lock obtain condition error</td>
</tr>
<tr>
<td>01</td>
<td>000004E6</td>
<td>ME</td>
<td>EBO</td>
<td>Shared memory lock obtain error</td>
</tr>
<tr>
<td>01</td>
<td>000004E7</td>
<td>ME</td>
<td>EBD</td>
<td>Shared memory lock destroy error</td>
</tr>
<tr>
<td>01</td>
<td>000004E8</td>
<td>ME</td>
<td>EBI</td>
<td>Shared memory lock initialize error</td>
</tr>
<tr>
<td>01</td>
<td>000004E9</td>
<td>ME</td>
<td>EFR</td>
<td>Mutex forced release</td>
</tr>
<tr>
<td>01</td>
<td>000004EA</td>
<td>ME</td>
<td>EFD</td>
<td>Mutex forced decrement</td>
</tr>
<tr>
<td>01</td>
<td>000004EB</td>
<td>ME</td>
<td>EDD</td>
<td>Mutex destroy failed (damage)</td>
</tr>
<tr>
<td>01</td>
<td>000004EC</td>
<td>ME</td>
<td>EDB</td>
<td>Mutex destroy failed (busy)</td>
</tr>
<tr>
<td>01</td>
<td>000004ED</td>
<td>ME</td>
<td>EIA</td>
<td>Mutex initialize failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000004EE</td>
<td>ME</td>
<td>EIS</td>
<td>Mutex initialize failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000004EF</td>
<td>ME</td>
<td>EF</td>
<td>Mutex release (forced by quiesce)</td>
</tr>
<tr>
<td>01</td>
<td>000004F0</td>
<td>ME</td>
<td>EP</td>
<td>Mutex program check</td>
</tr>
<tr>
<td>01</td>
<td>000004FA</td>
<td>ME</td>
<td>EDU</td>
<td>Mutex destroy failed (uninitialized)</td>
</tr>
<tr>
<td>01</td>
<td>000004FB</td>
<td>ME</td>
<td>EUI</td>
<td>Mutex uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000004FC</td>
<td>ME</td>
<td>EUO</td>
<td>Mutex unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000004FD</td>
<td>E</td>
<td>EO</td>
<td>Mutex already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000004FE</td>
<td>ME</td>
<td>EIN</td>
<td>Mutex initialization failed (duplicate)</td>
</tr>
<tr>
<td>01</td>
<td>00000508</td>
<td>CV</td>
<td>MR</td>
<td>CV release mutex</td>
</tr>
<tr>
<td>01</td>
<td>00000509</td>
<td>CV</td>
<td>MA</td>
<td>CV reacquire mutex</td>
</tr>
<tr>
<td>01</td>
<td>0000050A</td>
<td>CV</td>
<td>MW</td>
<td>CV mutex wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050B</td>
<td>CV</td>
<td>MAW</td>
<td>CV reacquire mutex after wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050C</td>
<td>CV</td>
<td>CW</td>
<td>CV condition wait</td>
</tr>
<tr>
<td>01</td>
<td>0000050D</td>
<td>CV</td>
<td>CTW</td>
<td>CV condition timeout</td>
</tr>
<tr>
<td>01</td>
<td>0000050E</td>
<td>CV</td>
<td>CWP</td>
<td>CV wait posted</td>
</tr>
<tr>
<td>01</td>
<td>0000050F</td>
<td>CV</td>
<td>CWI</td>
<td>CV wait interrupted</td>
</tr>
<tr>
<td>01</td>
<td>00000510</td>
<td>CV</td>
<td>CTO</td>
<td>CV wait timeout</td>
</tr>
<tr>
<td>01</td>
<td>00000511</td>
<td>CV</td>
<td>CSS</td>
<td>CV condition signal success</td>
</tr>
<tr>
<td>01</td>
<td>00000512</td>
<td>CV</td>
<td>CSM</td>
<td>CV condition signal miss</td>
</tr>
<tr>
<td>01</td>
<td>00000513</td>
<td>CV</td>
<td>CBS</td>
<td>CV condition broadcast success</td>
</tr>
<tr>
<td>01</td>
<td>00000514</td>
<td>CV</td>
<td>CBM</td>
<td>CV condition broadcast miss</td>
</tr>
<tr>
<td>01</td>
<td>00000515</td>
<td>CV</td>
<td>IN</td>
<td>CV initialize</td>
</tr>
<tr>
<td>01</td>
<td>00000516</td>
<td>CV</td>
<td>DS</td>
<td>CV destroy</td>
</tr>
<tr>
<td>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 SMC_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>00000529</td>
<td>CV</td>
<td>CSP</td>
<td>Call to SMC_POST</td>
</tr>
<tr>
<td>01</td>
<td>0000052A</td>
<td>CV</td>
<td>CSB</td>
<td>Call to SMC_POSTALL</td>
</tr>
<tr>
<td>01</td>
<td>0000052B</td>
<td>CV</td>
<td>CSW</td>
<td>Call to SMC_WAIT</td>
</tr>
<tr>
<td>01</td>
<td>0000052C</td>
<td>CV</td>
<td>DBM</td>
<td>Shared condition broadcast - miss</td>
</tr>
<tr>
<td>01</td>
<td>0000052D</td>
<td>CV</td>
<td>DBS</td>
<td>Shared condition broadcast - success</td>
</tr>
<tr>
<td>01</td>
<td>0000052E</td>
<td>CV</td>
<td>DDS</td>
<td>Destroy (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>0000052F</td>
<td>CV</td>
<td>DIN</td>
<td>Initialize (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000530</td>
<td>CV</td>
<td>DSM</td>
<td>Condition signal - miss (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000531</td>
<td>CV</td>
<td>DSS</td>
<td>Condition signal - success (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000532</td>
<td>CV</td>
<td>DWI</td>
<td>Wait interrupted (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000533</td>
<td>CV</td>
<td>DTO</td>
<td>Wait timeout (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>00000534</td>
<td>CV</td>
<td>DWP</td>
<td>Wait posted (shared CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006CB</td>
<td>CE</td>
<td>FBT</td>
<td>Error - Invalid system TOD (shared)</td>
</tr>
<tr>
<td>01</td>
<td>000006D1</td>
<td>CE</td>
<td>FRM</td>
<td>Error - Recursive mutex (shared)</td>
</tr>
<tr>
<td>01</td>
<td>000006D2</td>
<td>CE</td>
<td>FUO</td>
<td>Error - Shared mutex unowned</td>
</tr>
<tr>
<td>01</td>
<td>000006D3</td>
<td>CE</td>
<td>FDB</td>
<td>Error - Destroy failed (busy) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000006D4</td>
<td>CE</td>
<td>FDU</td>
<td>Error - Destroy failed (unitialized) (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 unialized</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>
</tbody>
</table>
**Table 32: LE=2 entry records (continued)**

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>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></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>
</tbody>
</table>
Table 32: LE=2 entry records (continued)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>000008E8</td>
<td>RE</td>
<td>EBI</td>
<td>Error - BPX1SLK(INIT) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E9</td>
<td>RE</td>
<td>EFR</td>
<td>Error - Forced release</td>
</tr>
<tr>
<td>01</td>
<td>000008EA</td>
<td>RE</td>
<td>EFD</td>
<td>Error - Forced decrement</td>
</tr>
<tr>
<td>01</td>
<td>000008ED</td>
<td>RE</td>
<td>EIA</td>
<td>Error - Initialization failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000008EE</td>
<td>RE</td>
<td>EIS</td>
<td>Error - Initialization failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000008EF</td>
<td>RE</td>
<td>EF</td>
<td>Error - Forced by quiesce</td>
</tr>
<tr>
<td>01</td>
<td>000008F0</td>
<td>RE</td>
<td>EP</td>
<td>Error - Program check</td>
</tr>
<tr>
<td>01</td>
<td>000008FB</td>
<td>RE</td>
<td>EUI</td>
<td>Error - Uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000008FC</td>
<td>RE</td>
<td>EUO</td>
<td>Error - Unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000008FD</td>
<td>RE</td>
<td>EO</td>
<td>Error - Already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008FE</td>
<td>RE</td>
<td>EIN</td>
<td>Error - Initialization failed (duplicate)</td>
</tr>
</tbody>
</table>

Table 33 on page 158 shows the format for the Mutex – Condition Variable – Latch entries in the trace table.

Table 33: Format of the mutex/CV/latch records.

<table>
<thead>
<tr>
<th>Record fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Source</td>
</tr>
<tr>
<td>Event</td>
</tr>
<tr>
<td>Object Addr</td>
</tr>
<tr>
<td>Name1</td>
</tr>
<tr>
<td>Name2</td>
</tr>
<tr>
<td>unused</td>
</tr>
</tbody>
</table>

### Class
Two character EBCDIC representation of the trace class.

- **LT**: Latch
- **LE**: Latch Exception
- **MX**: Mutex
- **ME**: Mutex Exception
- **CV**: Condition Variable
- **CE**: Condition Variable Exception

### Source
One character EBCDIC representation of the event.

- **C**: C/C++
- **S**: Sockets
- **Blank**: Blank character
**Event**

Two character EBCDIC representation of the event. See Table 32 on page 153.

**Object Addr**

Fullword address of the mutex object.

**Name 1**

Optional eight character field containing the name of the function or object to be recorded.

**Name 2**

Optional eight character field containing the name of the function or object to be recorded.

When LE=3 is specified

The trace table will include the records generated by both LE=1 and LE=2.

When LE=8 is specified

The trace table will contain only storage allocation records, as shown in Table 34 on page 159. Currently this is only supported by C/C++. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 195.

**Table 34: LE=8 entry records**

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000001</td>
<td>Storage allocation entry</td>
</tr>
<tr>
<td>03</td>
<td>00000001</td>
<td>Storage allocation exit</td>
</tr>
</tbody>
</table>

When LE=20 is specified

Table 35 on page 159 shows the C/C++ records that might be generated. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 195.

**Table 35: LE=20 entry records**

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000007</td>
<td>XPLINK calls non-XPLINK entry</td>
</tr>
<tr>
<td>03</td>
<td>00000008</td>
<td>non-XPLINK calls XPLINK entry</td>
</tr>
</tbody>
</table>

Sample dump for the trace table entry

The following sample shows an example of a dump of the trace table when you specify the LE=1 suboption (the library call/return trace).

Using Language Environment debugging facilities 159
Requesting a UNIX System Services syscall trace for debugging

Signal SIGTRACE can be sent to a process or process group to start or stop a trace of the z/OS UNIX System Services syscalls made by the application. The signal is implemented as a toggle. With the trace turned on, the z/OS UNIX System Services kernel gathers the syscall trace records for the targeted processes. A system dump of the user address space can be generated by sending signal SIGDUMP to the same processes to capture the trace output. See z/OS UNIX System Services Command Reference for more information about the SIGTRACE signal.
Part 2. Debugging language-specific routines

This part provides specific information for debugging applications written in C/C++, COBOL, Fortran, and PL/I. It also discusses techniques for debugging under CICS.
Chapter 4. Debugging C/C++ routines

This chapter provides specific information to help you debug applications that contain one or more C/C++ routines. It also provides information about debugging C/C++ applications compiled with XPLINK. It includes the following topics:

- Debugging C/C++ I/O routines
- Using C/C++ compiler listings
- Generating a Language Environment dump of a C/C++ routine
- Generating a Language Environment dump of a C/C++ routine with XPLINK
- Finding C/C++ information in a Language Environment dump
- Debugging example of C/C++ routines
- Debugging example of C/C++ routines with XPLINK

There are several debugging features that are unique to C/C++ routines. Before examining the C/C++ techniques to find errors, you might want to consider the following areas of potential problems:

- If you suspect that you are using uninitialized storage, you may want to use the STORAGE runtime option.
- If you are using the fetch() function, see z/OS XL C/C++ Programming Guide to ensure that you are creating the fetchable module correctly.
- If you are using DLLs, see z/OS XL C/C++ Programming Guide to ensure that you are using the DLL correctly.
- For non-System Programming C routines, ensure that the entry point of the load module is CEESTART.
- You should avoid:
  - Incorrect casting
  - Referencing an array element with a subscript outside the declared bounds
  - Copying a string to a target with a shorter length than the source string
  - Declaring but not initializing a pointer variable, or using a pointer to allocated storage that has already been freed

If a routine exception occurred and you need more information than the condition handler provided, run your routine with the following runtime options: TRAP(ON, NOSPIE) and TERMTHDACT(UAIMM). Setting these runtime options generates a system dump of the user address space of the original abend or program interrupt prior to the Language Environment condition manager processing the condition. After the system dump is taken by the operating system, the Language Environment condition manager continues processing.

Debugging C/C++ programs

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

Using the __amrc and __amrc2 structures to debug input/output

__amrc, a structure defined in stdio.h, can help you determine the cause of errors resulting from an I/O operation, because it contains diagnostic information (for example, the return code from a failed VSAM operation). There are two structures:

- __amrc (defined by type __amrc_type
- __amrc2 (defined by type __amrc2_type)

The __amrc2_type structure contains secondary information that C can provide.
Because any I/O function calls, such as `printf()`, can change the value of `__amrc` or `__amrc2`, make sure you save the contents into temporary structures of `__amrc_type` and `__amrc2_type` respectively, before dumping them.

Figure 36 on page 164 shows the structure as it appears in `stdio.h`.

```c
typedef struct __amrctype {
    union {
        long int __error;
        struct {
            unsigned short __syscode,
            __rc;
        } __abend;
        struct {
            unsigned char __fdbk_fill,
            __rc,
            __ftncd,
            __fdbk;
        } __feedback;
        struct {
            unsigned short __svc99_info,
            __svc99_error;
        } __alloc;
        union {
            void __code;
            struct {
                unsigned long __RBA;
                unsigned int __last_op;
                struct {
                    unsigned long __len_fill; /* __len + 4 */
                    unsigned long __len;
                    char __str[120];
                    unsigned long __parmr0;
                    unsigned long __parmr1;
                    unsigned long __fill2[2];
                    char __str2[64];
                } __msg;
            } __msg;
            #if __EDC_TARGET >= 0x22080000
                unsigned char __rplfdbwd[4];
            #endif
            #if __EDC_TARGET >= 0x41080000
                #ifdef __LP64
                    unsigned long __XRBA;
                #elif defined(__LL)
                    unsigned long long __XRBA;
                #else
                    unsigned int __XRBA1;
                    unsigned int __XRBA2;
                #endif
                unsigned char __amrc_noseek_to_seek;
                char __amrc_pad[23];
            #endif
        } __amrc_type;
    }
}
```

Figure 36: `__amrc` structure

Figure 37 on page 164 shows the `__amrc2` structure as it appears in `stdio.h`.

```c
struct {
    long int __error2;
    char __pad__error2[4];
    FILE *__fileptr;
    long int __reserved[6];
}
```

Figure 37: `__amrc2` structure
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.

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.

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.

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.

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

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.

A field containing a value that indicates the last I/O operation being performed by C/C++ at the time the error occurred. These values are shown in Table 36 on page 166.

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.

This field contains the reason for the switch from QSAM (noseek) to BSAM with NOTE and POINT macros requested (seek) by the XL C/C++ Runtime Library. This field is set when system-level I/O macro processing triggers an ABEND condition. The macro name values (defined in stdio.h) for this field are as follows:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>__AM_BSAM_NOSWITCH</td>
<td>No switch was made.</td>
</tr>
<tr>
<td>__AM_BSAM_UPDATE</td>
<td>The data set is open for update</td>
</tr>
<tr>
<td>__AM_BSAM_BSAMWRITE</td>
<td>The data set is already open for write (or update) in the same C process.</td>
</tr>
<tr>
<td>__AM_BSAM_FBS_APPEND</td>
<td>The data set is recfm=FBS and open for append</td>
</tr>
<tr>
<td>__AM_BSAM_LRECLX</td>
<td>The data set is recfm=LRECLX (used for VBS data sets where records span the largest blocksize allowed on the device)</td>
</tr>
<tr>
<td>__AM_BSAM_PARTITIONED_DIRECTORY</td>
<td>The data set is the directory for a regular or extended partitioned data set</td>
</tr>
<tr>
<td>__AM_BSAM_PARTITIONED_INDIRECT</td>
<td>The data set is a member of a partitioned data set, and the member name was not specified at allocation</td>
</tr>
</tbody>
</table>

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().

Debugging C/C++ routines 165
A secondary error code. For example, an unsuccessful rename or remove operation places its reason code here.

A pointer to the file that caused a SIGIOERR to be raised. Use an fldata() call to get the actual name of the file.

Reserved for future use.

The __last_op field is the most important of the __amrc fields. It defines the last I/O operation C/C++ was performing at the time of the I/O error. You should note that the structure is neither cleared nor set by non-I/O operations, so querying this field outside of a SIGIOERR handler should only be done immediately after I/O operations. Table 36 on page 166 lists __last_op values you could receive and where to look for further information.

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__IO_INIT</td>
<td>Will never be seen by SIGIOERR exit value given at initialization.</td>
</tr>
<tr>
<td>__BSAM_OPEN</td>
<td>Sets __error with return code from OS OPEN macro.</td>
</tr>
<tr>
<td>__BSAM_CLOSE</td>
<td>Sets __error with return code from OS CLOSE macro.</td>
</tr>
<tr>
<td>__BSAM_READ</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 66) filled in).</td>
</tr>
<tr>
<td>__BSAM_NOTE</td>
<td>NOTE returned 0 unexpectedly, no return code.</td>
</tr>
<tr>
<td>__BSAM_POINT</td>
<td>This will not appear as an error lastop.</td>
</tr>
<tr>
<td>__BSAM_WRITE</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 66) filled in).</td>
</tr>
<tr>
<td>__BSAM_CLOSE_T</td>
<td>Sets __error with return code from OS CLOSE TYPE=T.</td>
</tr>
<tr>
<td>__BSAM_BLDL</td>
<td>Sets __error with return code from OS BLDL macro.</td>
</tr>
<tr>
<td>__BSAM_STOW</td>
<td>Sets __error with return code from OS STOW macro.</td>
</tr>
<tr>
<td>__TGET_READ</td>
<td>Sets __error with return code from TSO TGET macro.</td>
</tr>
<tr>
<td>__TPUT_WRITE</td>
<td>Sets __error with return code from TSO TPUT macro.</td>
</tr>
<tr>
<td>__IO_DEVTYPE</td>
<td>Sets __error with return code from I/O DEVTYPE macro.</td>
</tr>
<tr>
<td>__IO_RDJFCB</td>
<td>Sets __error with return code from I/O RDJFCB macro.</td>
</tr>
<tr>
<td>__IO_TRKCALC</td>
<td>Sets __error with return code from I/O TRKCALC macro.</td>
</tr>
<tr>
<td>__IO_OBTAIN</td>
<td>Sets __error with return code from I/O CAMLST OBTAIN.</td>
</tr>
<tr>
<td>__IO_LOCATE</td>
<td>Sets __error with return code from I/O CAMLST LOCATE.</td>
</tr>
<tr>
<td>__IO_CATALOG</td>
<td>Sets __error with return code from I/O CAMLST CAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>__IO_UNCATALOG</td>
<td>Sets __error with return code from I/O CAMLST UNCAT. The associated macro is CATALOG.</td>
</tr>
<tr>
<td>__IO_RENAME</td>
<td>Sets __error with return code from I/O CAMLST RENAME.</td>
</tr>
<tr>
<td>__SVC99_ALLOC</td>
<td>Sets __alloc structure with info and error codes from SVC 99 allocation.</td>
</tr>
<tr>
<td>__SVC99_ALLOC_NEW</td>
<td>Sets __alloc structure with info and error codes from SVC 99 allocation of NEW file.</td>
</tr>
<tr>
<td>__SVC99_UNALLOC</td>
<td>Sets __unalloc structure with info and error codes from SVC 99 unallocation.</td>
</tr>
<tr>
<td>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__C_TRUNCATE</td>
<td>Set when C or C++ truncates output data. Usually, this is data written to a text file with no newline such that the record fills up to capacity and subsequent characters cannot be written. For a record I/O file this refers to an fwrite() writing more data than the record can hold. Truncation is always rightmost data. There is no return code.</td>
</tr>
<tr>
<td>__C_FCBCHECK</td>
<td>Set when C or C++ FCB is corrupted. This is due to a pointer corruption somewhere. File cannot be used after this.</td>
</tr>
<tr>
<td>__C_DBCS_TRUNCATE</td>
<td>This occurs when writing DBCS data to a text file and there is no room left in a physical record for anymore double byte characters. A new-line is not acceptable at this point. Truncation will continue to occur until an SI is written or the file position is moved. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_SO_TRUNCATE</td>
<td>This occurs when there is not enough room in a record to start any DBCS string or else when a redundant SO is written to the file before an SI. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_SI_TRUNCATE</td>
<td>This occurs only when there was not enough room to start a DBCS string and data was written anyways, with an SI to end it. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_UNEVEN</td>
<td>This occurs when an SI is written before the last double byte character is completed, thereby forcing C or C++ to fill in the last byte of the DBCS string with a padding byte X'FE'. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_CANNOT_EXTEND</td>
<td>This occurs when an attempt is made to extend a file that allows writing, but cannot be extended. Typically this is a member of a partitioned data set being opened for update.</td>
</tr>
<tr>
<td>__VSAM_OPEN_FAIL</td>
<td>Set when a low level VSAM OPEN fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_RRDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is RRDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_KSDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is KSDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS_PATH</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS PATH.</td>
</tr>
<tr>
<td>__VSAM_OPEN_KSDS_PATH</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is KSDS PATH.</td>
</tr>
<tr>
<td>__VSAM_MODCB</td>
<td>Set when a low level VSAM MODCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_TESTCB</td>
<td>Set when a low level VSAM TESTCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_SHOWCB</td>
<td>Set when a low level VSAM SHOWCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_GENCB</td>
<td>Set when a low level VSAM GENCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_GET</td>
<td>Set when the last op was a low level VSAM GET; if the GET fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_PUT</td>
<td>Set when the last op was a low level VSAM PUT; if the PUT fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_POINT</td>
<td>Set when the last op was a low level VSAM POINT; if the POINT fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__VSAM_ERASE</td>
<td>Set when the last op was a low level VSAM ERASE; if the ERASE fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_ENDREQ</td>
<td>Set when the last op was a low level VSAM ENDREQ; if the ENDREQ fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_CLOSE</td>
<td>Set when the last op was a low level VSAM CLOSE; if the CLOSE fails, sets __rc and __fdbk in the __amrc struct.</td>
</tr>
<tr>
<td>__QSAM_GET</td>
<td>__error is not set (if abend (errno == 92), __abend is set, otherwise if read error (errno == 66), look at __msg.</td>
</tr>
<tr>
<td>__QSAM_PUT</td>
<td>__error is not set (if abend (errno == 92), __abend is set, otherwise if write error (errno == 65), look at __msg.</td>
</tr>
<tr>
<td>__QSAM_TRUNC</td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td>__QSAM_FREEPOOL</td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td>__QSAM_CLOSE</td>
<td>Sets __error to result of OS CLOSE macro.</td>
</tr>
<tr>
<td>__QSAM_OPEN</td>
<td>Sets __error to result of OS OPEN macro.</td>
</tr>
<tr>
<td>__CMS_OPEN</td>
<td>Sets __error to result of FSOPEN.</td>
</tr>
<tr>
<td>__CMS_CLOSE</td>
<td>Sets __error to result of FSCLOSE.</td>
</tr>
<tr>
<td>__CMS_READ</td>
<td>Sets __error to result of FSREAD.</td>
</tr>
<tr>
<td>__CMS_WRITE</td>
<td>Sets __error to result of FSWRITE.</td>
</tr>
<tr>
<td>__CMS_STATE</td>
<td>Sets __error to result of FSSTATE.</td>
</tr>
<tr>
<td>__CMS_ERASE</td>
<td>Sets __error to result of FSERASE.</td>
</tr>
<tr>
<td>__CMS_RENAME</td>
<td>Sets __error to result of CMS RENAME command.</td>
</tr>
<tr>
<td>__CMS_EXTRACT</td>
<td>Sets __error to result of DMS EXTRACT call.</td>
</tr>
<tr>
<td>__CMS_LINERD</td>
<td>Sets __error to result of LINERD macro.</td>
</tr>
<tr>
<td>__CMS_LINEWRT</td>
<td>Sets __error to result of LINEWRT macro.</td>
</tr>
<tr>
<td>__CMS_QUERY</td>
<td>__error is not set.</td>
</tr>
<tr>
<td>__HSP_CREATE</td>
<td>Indicates last op was a DSPSERV CREATE to create a hiperspace for a hiperspace memory file. If CREATE fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__HSP_DELETE</td>
<td>Indicates last op was a DSPSERV DELETE to delete a hiperspace for a hiperspace memory file during termination. If DELETE fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__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>
</tbody>
</table>
### Table 36: __last_op values and diagnosis information (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__LFS_OPEN</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_CLOSE</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_READ</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_WRITE</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_LSEEK</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_FSTAT</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
</tbody>
</table>

### Using file I/O tracing to debug C/C++ file I/O problems

You can use file I/O tracing to debug C/C++ file I/O problems. For more information, see Debugging I/O programs in z/OS XL C/C++ Programming Guide.

### Displaying an error message with the perror() function

To find a failing routine, check the return code of all function calls. After you have found the failing routine, use the perror() function after the routine to display the error message. perror() displays the string that you pass to it and an error message corresponding to the value of errno. perror() writes to the standard error stream (stderr). Figure 38 on page 169 is an example of a routine using perror().

By default, the errno2 value will be appended to the end of the perror() string. If you do not want the errno2 value appended to the perror() string, set the _EDC_ADD_ERRNO2 environment variable to 0.

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

Figure 38: Example of a routine using perror()

### Using __errno2() to diagnose application problems

Use the __errno2() function when diagnosing problems in an application program. This function enables z/OS XL C/C++ application programs to access additional diagnostic information, errno2 (errnojr), associated with errno. The __errno2 may be set by the z/OS XL C/C++ runtime library, z/OS UNIX callable services, or other callable services. The errno2 is intended for diagnostic display purposes only and is not a programming interface.
**Note:** Not all functions set errno2 when errno is set. In the cases where errno2 is not set, the __errno2() function may return a residual value. You may use the __err2ad() function to clear errno2 to reduce the possibility of a residual value being returned.

Figure 39 on page 170 is an example of a routine using __errno2().

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

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

**Figure 39: Example of a routine using __errno2()**

Figure 40 on page 170 shows the output from the sample routine in Figure 39 on page 170.

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

**Figure 40: Sample output of a routine using __errno2()**

Figure 41 on page 170 is an example of a routine using the environment variable _EDC_ADD_ERRNO2.

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

int main(void) {
    FILE *fp;
    /* do NOT add errno2 to perror message */
    setenv("_EDC_ADD_ERRNO2", "0", 1);
    fp = fopen("testfile.dat", "r");
    if (fp == NULL) {
        perror("fopen() failed");
    } return 0;
}
```

**Figure 41: Example of a routine using _EDC_ADD_ERRNO2**

Figure 42 on page 170 shows the sample output from the routine in Figure 41 on page 170.

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

**Figure 42: Sample output of a routine using _EDC_ADD_ERRNO2**

Figure 43 on page 171 is an example of a routine using __err2ad() in combination with __errno2().
```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(void) {
    FILE *f;
    setenv("_EDC_ADD_ERRNO2", "0", 1);
    f = fopen("testfile.dat", "r");
    if (f == NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    } /* reset errno2 to zero */
    *__err2ad() = 0x0;
    printf("__errno2 = %08x\n", __errno2());
    f = fopen("testfile.dat", "r");
    if (f == NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    } return 0;
}
```

Figure 43: Example of a routine using __err2ad() in combination with __errno2()

Figure 44 on page 171 shows the sample output from the routine shown in Figure 43 on page 171.

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

Figure 44: Sample output of routine using __err2ad() in combination with __errno2()

For more information about _EDC_ADD_ERRNO2, see z/OS XL C/C++ Programming Guide.
For more information about __errno2() and __err2ad(), see z/OS XL C/C++ Runtime Library Reference.

Diagnosing DLL problems

Use the _EDC_DLL_DIAG environment variable to diagnose DLL problems. For more information, see z/OS XL C/C++ Programming Guide

You can also see the diagnosis output in CEEDUMP and Verbexit LEDATA reports. For more information, see “Using the DLL failure control block” on page 79.

Using C/C++ listings

For a detailed description of available listings, see z/OS XL C/C++ User’s Guide.

Finding variables

You can determine the value of a variable in the routine at the point of interrupt by using the compiled code listing as a guide to its address, then finding this address in the Language Environment dump. The method you use depends on the storage class of variable.

This method is generally used when no symbolic variables have been dumped (by using the TEST compiler option).
It is possible for the routine to be interrupted before the value of the variable is placed in the location provided for it. This can explain unexpected values in the dump.

Steps for finding automatic variables

Perform the following steps to find automatic variables in the Language Environment dump:

1. Identify the start of the stack frame. If a dump has been taken, each stack frame is dumped. The stack frames can be cross-referenced to the function name in the traceback.
2. Determine the value of the base register (in this example, GPR13) in the Saved Registers section for the function you are interested in.
3. Find the offset of the variable (which is given in decimal) in the storage offset listing.

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

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

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

Locating the Writable Static Area (WSA)

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

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

Note:

1. Address calculation for static and external variables uses the static storage area as a base address with 1 or more offsets added to this address.
2. The storage associated with these CSECTs is not dumped when an exception occurs. It is dumped when 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 178.
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 172. For this procedure's example, assume that the address of writable static is X'02D66E40'.

Perform the following steps to find RENT static variables:

1. Find the offset of @STATIC (associated with the file where the static variable is located) in the Writable Static Map section of the prelinker map. Figure 45 on page 173 shows an example; in this Writable Static Map section of a prelinker map, the offset is X'58'.

2. Add the offset to the WSA to get the base address of static variables, as shown.

   X'02D66E40' + X'58' = X'2D66E98'

3. Find the offset of the static variable in the partial storage offset compiler listing. In the following example, the offset is 96 (X'60').

   sa0 66-0:66 Class = static, Location = WSA + @STATIC + 96, Length = 4

4. Add the offset of the static variable in the partial storage offset compiler listing (found in step 3) to the base address of static variables (calculated in step 2).

   X'2D66E98' + X'60' = X'2D66EF8'

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

Figure 46 on page 174 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 172. For this procedure's example, the address of writable static is '02D66E40'.

Perform the following steps to find external RENT variables:

1. Find the offset of the external variable in the Prelinker Writable Static Map. In the example shown in Figure 47 on page 174, the offset for DFHEIPTR is '28'.

```
|                         Writable Static Map                          |
|========================================================================|
| OFFSET    LENGTH  FILE ID  INPUT NAME                  |
| 0         1   00001   DFHC0011                          |
| 4         1   00001   DFHC0010                          |
| 8         2   00001   DFHDUMMY                           |
| C         2   00001   DFHB0025                           |
| 10        2   00001   DFHB0024                           |
| 14        2   00001   DFHB0023                           |
| 18        2   00001   DFHB0022                           |
| 1C        2   00001   DFHB0021                           |
| 20        2   00001   DFHB0020                           |
| 24        2   00001   DFHEIB0                            |
| 28        4   00001   DFHEIPTR                           |
| 2C        4   00001   DFHCP011                           |
| 30        4   00001   DFHCP010                           |
| 34        4   00001   DFHBP025                           |
| 38        4   00001   DFHBP024                           |
| 3C        4   00001   DFHBP023                           |
| 40        4   00001   DFHBP022                           |
| 44        4   00001   DFHBP021                           |
| 48        4   00001   DFHBP020                           |
| 4C        4   00001   DFHEIBC                            |
| 50        4   00001   DFHEID0                            |
| 54        4   00001   DFHLDSVR                            |
| 58        420   00001   @STATIC                           |
```

Figure 47: Writable static map produced by prelinker

2. Add the offset of the external variable to the address of writable static, as shown below.

```
X'02D66E40' + X'28' = X'2D66E68'
```

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

Steps for finding NORENT static variables

Before you begin: You need to know the name and address of the static storage area. To find this information see “Steps for finding the static storage area” on page 172. For this procedure's example, the static storage area is called STATSTOR and has an address of '02D66E40'.

Perform the following steps to find external RENT variables:

1. Find the offset of the static variable in the partial storage offset compiler listing. As shown in the following example, the offset is '96 (X'60').

```
sa0  66-0:66  Class = static,  Location = STATSTOR +96,  Length = 4
```
2. Add the offset to the base address of static variables, as shown in the following example:

\[ \text{X}'2D66E40' + \text{X}'60' = \text{X}'2D66EA0' \]

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

Figure 48 on page 175 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 172. For this procedure’s example, the address of the external variable CSECT is X'02D66E40'.

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

**Steps for finding the C/370 parameter list**

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

1. Identify the address of the start of the parameter list. A pointer to the parameter list is passed to the called function in register 1. This is the address of the start of the parameter list. Figure 49 on page 175 shows an example code for the parameter variable.

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

*Figure 49: Example code for parameter variable*

Parameters `ppx` and `pp0` correspond to copies of `a1` and `a2` in the stack frame belonging to `func0`.

2. Use the address of the start of the parameter list to find the register and offset in the partial storage offset listing. As shown in the following example, the offset is 4 (X'4') from register 1.

| pp0 | 62-0:62 | Class = parameter, Location = 4(x1), Length = 4 |

3. Determine the value of GPR1 in the Saved Registers section for the function that called the function you are interested in.

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

When you are done, the contents of the variable can then be read in the DSA frame section corresponding to the function the parameter was passed from.
Steps for finding the C++ parameter list

Before you begin: To locate C++ functions with extern C attributes, see “Steps for finding the C/370 parameter list” on page 175.

Perform the following steps to find the C++ parameter list:

1. Identify the address of the start of the parameter list. A pointer to the parameter list is passed to the called function in register 1. This is the address of the start of the parameter list. Figure 50 on page 176 shows an example code for the parameter variable.

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

   *Figure 50: Example code for parameter variable*

   Parameters ppx and pp0 correspond to copies of a1 and a2 in the stack frame belonging to func1.

2. Locate the value of the base register in the Saved Registers section of the function you are interested in.

3. Find the offset of the static variable in the partial storage offset compiler listing, as shown in Figure 51 on page 176.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Offset</th>
<th>Class</th>
<th>Location</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppx</td>
<td>62-0:62</td>
<td>parameter</td>
<td>188(r13)</td>
<td>4</td>
</tr>
<tr>
<td>pp0</td>
<td>62-0:62</td>
<td>parameter</td>
<td>192(r13)</td>
<td>4</td>
</tr>
</tbody>
</table>

   *Figure 51: Partial storage offset listing*

4. Add the value of the base register to the offset.

5. Locate the parameter.

Restriction: When OPTIMIZE is on, the parameter value might never be stored, since the first few parameters might be passed in registers and there might be no need to save them.

Steps for finding members of aggregates

You can define aggregates in any of the storage classes or pass them as parameters to a called function. The first step is to find the start of the aggregate. You can compute the start of the aggregate as described in previous sections, depending on the type of aggregate used.

The aggregate map provided for each declaration in a routine can further assist in finding the offset of a specific variable within an aggregate. Structure maps are generated using the AGGREGATE compiler option. Figure 52 on page 176 shows an example of a static aggregate.

   ```c
   static struct {
   short int ss01;
   char ss02[56];
   int ss03;
   int ss89;
   } ss0;
   ```

   *Figure 52: Example code for structure variable*

   Figure 53 on page 177 shows an example aggregate map.
Figure 53: Example of aggregate map

Assume the structure has been compiled as RENT. To find the value of variable sz0[0]:

1. Find the address of the writable static. For this example the address of writable static is X'02D66E40'.
2. Find the offset of @STATIC in the Writable Static Map. In this example, the offset is X'58'. Add this offset to the address of writable static. The result is X'2D66E98' (X'02D66E40' + X'58'). Figure 54 on page 177 shows the Writable Static Map produced by the prelinker.

3. Find the offset of the static variable in the storage offset listing. The offset is 96 (X'60'). The following is an example of a partial storage offset listing.

4. Find the offset of sz0 in the Aggregate Map, shown in Figure 53 on page 177. 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.

Figure 54: Writable static map produced by prelinker
Finding the timestamp

The timestamp is in the compile unit block. The address for the compile unit block is located at eight bytes past the function entry point. The compile unit block is the same for all functions in the same compilation. The fourth word of the compile unit block points to the timestamp. The timestamp is 16 bytes long and has the following format:

YYYYMMDDHHMMSSSS

Generating a Language Environment dump of a C/C++ routine

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

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

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

See the z/OS XL C/C++ Runtime Library Reference for more details about the syntax of these functions.

cdump()

If your routine is running under z/OS or CICS, you can generate useful diagnostic information by using the cdump() function. cdump() produces a main storage dump with the activation stack. This is equivalent to calling CEE3DMP with the option string:

TRACEBACK BLOCKS VARIABLES FILES STORAGE STACKFRAME(ALL) CONDITION ENTRY

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

The output of the dump is directed to the CEESNAP data set. The DD definition for CEESNAP is as follows:

//CEESNAP DD SYSOUT= *

If the data set is not defined, or is not usable for any reason, cdump() returns a failure code of 1. This occurs even if the call to CEE3DMP is successful. If the SNAP is not successful, the CEE3DMP DUMP file displays the following message:

Snap was unsuccessful

If the SNAP is successful, CEE3DMP displays the following message, where nnn corresponds to the SNAP identifier described above. An unsuccessful SNAP does not result in an incrementation of the identifier.

Snap was successful; snap ID = nnn

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

Support for SNAP dumps using the _cdump function is provided only under z/OS and z/VM. SNAP dumps are not supported under CICS; no SNAP is produced in this environment. A successful SNAP results in a
large quantity of output. A routine calling cdump() under CICS receives a return code of 0 if the ensuing call to CEE3DMP is successful. In addition to a SNAP dump, a Language Environment formatted dump is also taken.

csnap()

The csnap() function produces a condensed storage dump. csnap() is equivalent to calling CEE3DMP with the option string:

```
TRACEBACK FILES BLOCKS VARIABLES NOSTORAGE STACKFRAME(ALL) CONDITION ENTRY
```

ctrace()

The ctrace() function produces a traceback and includes the offset addresses from which the calls were made. ctrace() is equivalent to calling CEE3DMP with the option string:

```
TRACEBACK NOFILES NOBLOCKS NOVARIABLES NOSTORAGE STACKFRAME(ALL) NOCONDITION NOENTRY
```

Sample C routine that calls cdump()

The code example below shows a sample C routine that uses the cdump function to generate a dump. The sample here shows the dump output.

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void hsigfpe(int);
void hsigterm(int);
void atf1(void);

typedef int (*FuncPtr_T)(void);

int st1    = 99;
int st2    = 255;
int xcount = 0;

int main(void) {
/*
 * 1) Open multiple files
 * 2) Register 2 signals
 * 3) Register 1 atexit function
 * 4) Fetch and execute a module
 */

    FuncPtr_T fetchPtr;
    FILE* fp1;
    FILE* fp2;
    int rc;
    fp1 = fopen("myfile.data", "w");
    if (!fp1) {
        perror("Could not open myfile.data for write");
        exit(101);
    }
    fprintf(fp1, "record 1\n");
    fprintf(fp1, "record 2\n");
    fprintf(fp1, "record 3\n");
    fp2 = fopen("memory.data", "wb,type=memory");
    if (!fp2) {
        perror("Could not open memory.data for write");
```
exit(102);
}

fprintf(fp2, "some data");
fprintf(fp2, "some more data");
fprintf(fp2, "even more data");
signal(SIGFPE, hsigfpe);
signal(SIGTERM, hsigterm);

rc = atexit(atf1);
if (rc) {
    fprintf(stderr, "Failed on registration of atexit function atf1\n");
    exit(103);
}

fetchPtr = (FuncPtr_T) fetch("MODULE1");
if (!fetchPtr) {
    fprintf(stderr, "Failed to fetch MODULE1\n");
    exit(104);
}
fetchPtr();
return(0);
}

void hsigfpe(int sig) {
    ++st1;
    return;
}

void hsigterm(int sig) {
    ++st2;
    return;
}

void atf1() {
    ++xcount;
}

Figure 55 on page 180 shows a fetched C module.

#include <ctest.h>
#pragma linkage(func1, fetchable)
int func1(void) {
    cdump("This is a sample dump");
    return(0);
}

Figure 55: Fetched module for C routine

Sample C++ routine that generates a Language Environment dump

Figure 56 on page 181 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 56: Example C++ routine with protection exception generating a dump*

Figure 57 on page 181 shows the template file stack.c

```cpp
#ifndef __STACK__
#define __STACK__
#include "stack.h"
#endif

template <class T> T Stack<T>::pop() {
    T value = head->value;
    head = head->next;
    return(value);
}

template <class T> void Stack<T>::push(T value) {
    Node* newNode  = new Node;
    newNode->value = value;
    newNode->next  = head;
    head = newNode;
}
```

*Figure 57: Template file STACK.C*

Figure 58 on page 181 shows the header file stack.h.

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

*Figure 58: Header file STACK.H*

**Sample Language Environment dump with C/C++-specific information**

The sample dump below was produced by compiling the routine in this sample with the TEST(SYM) compiler option, then running it. Notice the sequence of calls in the traceback section - EDCZMINV is the C-C++ management module that invokes main and @@FECBMODULE1 fetches the user-defined function func1, which in turn calls the library routine __cdump.
If source code is compiled with the GONUMBER or TEST compile option, statement numbers are shown in the traceback. If source code is compiled with the TEST (SYM) compile option, variables and their associated type and value are dumped out. Note that the high half of register 14 at entry to CEE3DMP is not available and is shown in the dump as *********. For more information about C/C++-specific information contained in a dump, see “Finding C/C++ information in a Language Environment dump” on page 186.
Debugging C/C++ routines

183
fldata FOR FILE: HEALY.MEMORY.DATA
__recfmF: 1........ 1
__recfmV: 1........ 0
__recfmS: 1........ 0
__recfmBlk: 1...... 0
__recfmASA: 1...... 0
__recfmM: 1........ 0
__recfmPO: 1....... 0
__dsorgPDSmem: 1... 0
__dsorgPS: 1....... 0
__dsorgConcat: 1... 0
__dsorgMem: 1...... 1
__dsorgHiper: 1.... 0
__dsorgTemp: 1..... 0
__dsorgHFS: 1...... 0
__openmode: 2...... 1
__modeflag: 4...... 2
__blksize......... 1024
__maxreclen....... 1024
__noseek_to_seek.. 0(0)
function pointer... A0901FF8    WSA address... 20FC7038
function name... atf1
__dsname.......... HEALY.MEMORY.DATA
FILE pointer........ 20FF5AFC
Buffer at current file position: 20FF5EE8
Saved Buffer....... NULL
File Control Block: 20FF4038
__blksize......... 1024
__maxreclen....... 1024
__noseek_to_seek.. 0(0)
__openmode: 2...... 1
__modeflag: 4...... 2
__access_method... 0(0)
__seek_to_seek... 0(0)
__dsname........... HEALY.MEMORY.DATA
FILE pointer......... 20FF5AFC
Buffer at current file position: 20FF5EE8
fldata 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
ddbname.............. SYS00001
Buffer at current file position: 20FF42E8

+000000 20FF4024  00280000 000C0000 99858396 998440F1  000C0000 99858396 998440F2 000C0000  |........record 1....record 2....|
+000020 20FF4308  99858396 998440F3 00040000 00000000  00000000 00000000 00000000 00000000  |record 3........................|
+000040 20FF4328  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  |................................|
+000060 20FF4348 - +0003FF 20FF46E7             same as above
Saved Buffer........ NULL
Write Data Control Block: 00015020
+000000 00015020  20FF42A8 00000000 000B0003 00E02026  002FE5A2 00000001 00004000 00006D40  |...y..............Vs...... ..._ |
+000020 00015040  86000000 500157D0 00CC2424 008CED84  12B745B8 00B7E220 0A000000 00001800  |f...&..........d......S.........|
+000040 00015060  30013030 00006DB0 01C45470 00C45470  00000404 00C45A08 90ECD00C 18BF58A0  |......_..D...D.......D!.........|
read/update DCB..... NULL
Write Data Control Block Extension: 20FF42A8
+000000 20FF42A8  C4C3C2C5 00380000 00015020 00000000  C0880000 00000000 00000000 00000000  |DCBE......&......h..............|
+000020 20FF42C8  00000000 00000000 20B74AB2 20B74A54  00000000 00000100 80001810 20FF4000  |.............................. .|
read/update DCBE.... NULL
Job File Control Block: 000157E8
+000000 000157E8  C8C5C1D3 E84BD4E8 C6C9D3C5 4BC4C1E3  C1404040 40404040 40404040 40404040  |HEALY.MYFILE.DATA               |
+000020 00015808  40404040 40404040 40404040 40404040  40404040 80000000 00000000 00000000  |                    ............|
+000040 00015828  00000200 00000000 00000000 00000000  6B000A00 00000040 00000000 00000000  |................,...... ........|
+000060 00015848  00000000 00000000 00000000 00000000  00000000 0001E2D3 F8C2F1F3 40404040  |......................SL8B13    |
+000080 00015868  40404040 40404040 40404040 40404040  40404040 000003AF 00000000 00000000  |                    ............|
+0000A0 00015888  00000000 00000000 00000000 00000100 80000038 00015000 000158A0 20FF42E8  |......................&........Y|

[8] amrc_type structure: 20FCDAB8
+000000 20FCDAB8  00000000 00000000 00000007 00000000  00000000 00000000 00000000 00000000  |................................|
+000020 20FCDAD8  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  |................................|
+000040 20FCDAF8 - +0000FF 20FCDBB7             same as above
amrc __code union fields
__error................. 0(0)
__abend.__syscode....... 0(0)
__abend.__rc............ 0(0)
__feedback.rc........... 0(0)
__feedback.__ftncd...... 0(0)
__feedback.__fdbk....... 0(0)
__alloc.__svc99_info.... 0(0)
__alloc.__svc99_error... 0(0)
__RBA............... 0(0)
__last_op........... 7(7)
__msg__str......... NULL
__msg__parmr0...... 0(0)
__msg__parmr1...... 0(0)
__msg__str2........ NULL
__rplfdbwd.............. 0x00000000
__amrc_noseek_to_seek... 0(0)
__XRBA.................. 0(0)

[9] error information :
errno information :
Debugging C/C++ routines 185
Additional Language Specific Information:
Finding C/C++ information in a Language Environment dump

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

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

Each of the unique COBOL sections of the Language Environment dump are described in Table 38 on page 186.

<table>
<thead>
<tr>
<th>Table 38: Contents of the COBOL sections of Language Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Section Number and Heading</strong></td>
</tr>
<tr>
<td>[1] Storage for Active Routines</td>
</tr>
</tbody>
</table>
| [2] Control Blocks Associated with the Active Thread | Contains the following information:  
\begin{itemize}
  \item Fields from the CAA  
  \item Fields specific from the CTHD and CEDB  
  \item Signal information
\end{itemize} |
| [2A] C/C++ CAA Fields                             | Contains several fields that the C/C++ programmer can use to find information about the runtime environment. For each COBOL program, there is a C-C++ Specific Thread area and a C-C++ Specific Enclave area. |
| [2B] C-C++ Specific CAA                          | The C-C++ specific CAA fields that are of interest to users are described below.  
\begin{itemize}
  \item **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.  
  \item **Memory file control block**  
  You can use the memory file control block (MFCB) to locate additional information about memory files. This control block resides at the COBOL thread level. For more information about the MFCB, see “Memory file control block” on page 188.  
  \item **Open FCB chain**  
  A pointer to the start of a linked list of open file control blocks (FCBs). For more information about FCBs, see File Control Block Information. |
| [3] Signal Information                           | When the POSIX(OFF) runtime option is specified, signal information is provided in the dump to aid you in debugging. For each signal that is disabled with 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. |
<p>| [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. |</p>
<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [5] atexit() Information  | Lists the functions registered with the `atexit()` function that would be run at normal termination. The functions are listed in chronological order of registration.  
If you compile a C routine as NORENT, the WSA address is not available (N/A). For more information about the `atexit()` function, see [z/OS XL C/C++ Runtime Library Reference](http://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary). |
| [6] fetch() Information   | Shows information about modules that you have dynamically loaded using `fetch()`. For each module that was fetched, the `fetch()` pointer and the function pointer are included.  
```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](http://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary). |
| [7] File Control Block Information | Includes the file control block (FCB) information for each C/C++ file. The FCB contains file status and attributes for files open during C/C++ active routines. You can use this information to find the data set or file name. The FCB is a handle that points to the following file information, which is displayed when applicable, for the file:  
- Access method control block (ACB) address  
- Data control block (DCB) address  
- Data control block extension (DCBE) address  
- Job file control block (JFCB) address  
- RPL address  
- Current buffer address  
- Saved buffer address  
- `ddname`  
Not all FCB fields are always filled in. For example, RPLs are used only for VSAM data sets. The `ddname` field contains blanks if it is not used.  
The save block buffer represents auxiliary buffers that are used to save the contents of the main buffers. Such saving occurs only when a reposition is performed and there is new data; for example, an incomplete text record or an incomplete fixed-block standard (FBS) block in the buffers that cannot be flushed out of the system.  
Because the main buffers represent the current position in the file, while the save buffers merely indicate a save has occurred, check the save buffers only if data appears to be missing from the external device and is not found in the main buffers. Also, do not infer that the presence of save buffers means that data present there belongs at the end of the file. (The buffers remain, even when the data is eventually written.)  
For information about the job file control block, see [z/OS MVS Data Areas in the z/OS Internet library](http://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary). |
| [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 163 and to [z/OS XL C/C++ Programming Guide](http://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary). |
| [9] Errno Information      | Shows the thread ID of the thread that generated the dump and the settings of the errno and errnojr variables for that thread. Both the errno and the errnojr variables contain the return code of the last failing z/OS UNIX system service call. These variables provide z/OS UNIX application programs access to diagnostic information returned from an underlying z/OS UNIX callable service. For more information on these return and reason codes, see [z/OS UNIX System Services Messages and Codes](http://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary). |
Memory file control block

This section of the dump holds the following memory file control block information for each memory file the routine uses. A sample memory file control block is shown in Figure 59 on page 188.

Memory file name

The name assigned to this memory file.

First memory data space

A dump of the first 1K maximum of actual user data associated with this memory file.

---

Figure 59: Memory file control block

---

Additional Floating-Point registers

The Language Environment dump formats Additional Floating Point (AFP) registers and Floating Point Control (FPC) registers when the AFP suboption of the FLOAT XL C/C++ compiler option is specified and the registers are needed. These floating-point registers are displayed in three sections of the CEEDUMP: Registers on Entry to CEE3DMP; Parameters, Registers, and Variables; and Condition Information for Active Routines. Samples of each section are given. For information on the FLOAT XL C/C++ compiler option, see z/OS XL C/C++ User's Guide.

Registers on entry to CEE3DMP

This section of the Language Environment dump displays the sixteen floating-point registers. Figure 60 on page 188 shows sample output. Note that the high half of general purpose register 14 at entry to CEE3DMP is not available and is shown in the dump as ********.

---

Figure 60: Registers on entry to CEE3DMP

---

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
Dashes are displayed in the registers when the register values are not saved. A sample output is shown.

### Parameters, Registers, and Variables for Active Routines

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR0</td>
<td>183F6CC0</td>
</tr>
<tr>
<td>GPR1</td>
<td>00021278</td>
</tr>
<tr>
<td>GPR2</td>
<td>183F6870</td>
</tr>
<tr>
<td>GPR3</td>
<td>17F01DC2</td>
</tr>
<tr>
<td>GPR4</td>
<td>00009FF8</td>
</tr>
<tr>
<td>GPR5</td>
<td>183F6968</td>
</tr>
<tr>
<td>GPR6</td>
<td>17F02408</td>
</tr>
<tr>
<td>GPR7</td>
<td>000012EC</td>
</tr>
<tr>
<td>GPR8</td>
<td>000212F0</td>
</tr>
<tr>
<td>GPR9</td>
<td>80000000</td>
</tr>
<tr>
<td>GPR10</td>
<td>98125022</td>
</tr>
<tr>
<td>GPR11</td>
<td>80007F98</td>
</tr>
<tr>
<td>GPR12</td>
<td>00015920</td>
</tr>
<tr>
<td>GPR13</td>
<td>000213B0</td>
</tr>
<tr>
<td>GPR14</td>
<td>97F01E1E</td>
</tr>
<tr>
<td>GPR15</td>
<td>0000002F</td>
</tr>
</tbody>
</table>

### Condition Information for Active Routines

This section of the Language Environment dump displays the floating-point registers when they are saved in the machine state; Figure 62 on page 189 shows sample output.

### Vector registers

The Language Environment dump formats vector registers when the vector registers are needed. These vector registers are displayed in three sections of the CEEDUMP: Registers on Entry to CEE3DMP; Parameters, Registers, and Variables; and Condition Information for Active Routines. Samples of each section are given.

### Registers on entry to CEE3DMP

This section of the Language Environment dump displays the 32 vector registers. Figure 63 on page 190 shows sample output.
Figure 63: Registers on entry to CEE3DMP

Parameters, registers, and variables for active routines

This section of the Language Environment dump displays the non-volatile vector registers that are saved in the stack frame. The registers are only displayed if the program owning the stack frame saved them. Asterisks are displayed in the registers when the register values are not saved. A sample output is shown.

Figure 64: Parameters, registers, and variables for active routines

Condition information for active routines

This section of the Language Environment dump displays the vector registers when they are saved in the machine state; Figure 65 on page 191 shows sample output.
Condition Information for Active Routines
Condition Information for (DSA address 25F27230)
CIB Address: 25F28560
Current Condition:
CEE3224S The system detected an IEEE division-by-zero exception.
Location:
Program Unit: Entry: goo Statement: 78 Offset: +000000BA
Machine State:
ILC..... 0004 Interruption Code..... 0007
PSW..... 078D0400 A570093C
GPR0..... 00000000_183F6CC0 GPR1..... 00000000_00021278 GPR2..... 00000000_183F6870 GPR3..... 00000000_0000122C
GPR4..... 00000000_0000012F GPR5..... 00000000_183F6968 GPR6..... 00000000_17F02408 GPR7..... 00000000_000212EC
GPR8..... 00000000_000212F0 GPR9..... 00000000_80000000 GPR10.... 00000000_98125022 GPR11.... 00000000_80007F98
GPR12.... 00000000_00015920 GPR13.... 00000000_000213B0 GPR14.... 00000000_97F01E1E GPR15.... 00000000_0000002F
FPC...... 40084000
FPR0..... 40260000 00000000 FPR1..... 41086A00 00000000
FPR2..... 00000000 00000000 FPR3..... 3F8CAC08 3126E979
FPR4..... 3FF33333 33333333 FPR5..... 40C19400 00000000
FPR6..... 3F661E4F 765FD8AE FPR7..... 3FF06666 66666666
FPR8..... 3FF33333 33333333 FPR9..... 00000000 00000000
FPR10.... 3FF33333 33333333 FPR11.... 00000000 00000000
FPR12.... 40260000 00000000 FPR13.... 00000000 00000000
FPR14.... 40220000 00000000 FPR15.... 00000000 00000000
VR0...... 40260000 00000000 00000000 33330000 VR1...... 41086A00 00000000 01000000 33330000
VR2...... 00000000 00000000 02000000 33330000 VR3...... 3F8CAC08 3126E979 03000000 33330000
VR4...... 3F331313 33331333 33330000 VR5...... 40C19400 00000000 05000000 33330000
VR6..... 3F661E4F 765FD8AE 06000000 33330000 VR7..... 3FF06666 66666666 07000000 33330000
VR8..... 3F331313 33331333 33330000 VR9..... 00000000 00000000 08000000 33330000
VR10.... 3F331313 33331333 33330000 VR11.... 00000000 00000000 09000000 33330000
VR12.... 40260000 00000000 0A000000 33330000 VR13.... 00000000 00000000 0B000000 33330000
VR14.... 40220000 00000000 0C000000 33330000 VR15.... 00000000 00000000 0D000000 33330000
VR16..... 10080000 33330000 0E000000 33330000 VR17.... 11000000 33330000 0F000000 33330000
VR18..... 12000000 33330000 10000000 33330000 VR19.... 13000000 33330000 11000000 33330000
VR20..... 14000000 33330000 12000000 33330000 VR21.... 15000000 33330000 13000000 33330000
VR22..... 16000000 33330000 14000000 33330000 VR23.... 17000000 33330000 15000000 33330000
VR24..... 18000000 33330000 16000000 33330000 VR25.... 19000000 33330000 17000000 33330000
VR26..... 1A000000 33330000 18000000 33330000 VR27.... 1B000000 33330000 19000000 33330000
VR28..... 1C000000 33330000 1A000000 33330000 VR29.... 1D000000 33330000 1B000000 33330000
VR30..... 1E000000 33330000 1C000000 33330000 VR31.... 1F000000 33330000 1D000000 33330000

Storage dump near condition, beginning at location: 25700928
&Lv;......

Figure 65: Condition information for active routines

Sample Language Environment dump with XPLINK-specific information

The programs tranmain (Figure 66 on page 192) and trandll (Figure 67 on page 192) were used to produce a Language Environment dump. The Language Environment dump produced by running these program is shown in “Example dump of calling between XPLINK and non-XPLINK programs” on page 192. 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. Explanations for some of the sections are in “Finding XPLINK information in a Language Environment dump” on page 194.
```c
#include <stdio.h>

#pragma export(tran2)
int tran2(int parm1,int parm2,int parm3,long double parm4,int parm5) {
  int retval;
  printf("Tran2: Call Tran3\n");
  retval = tran3(parm1,parm2,parm3,parm4,parm5);
  printf("Tran2: Return value from Tran3 = %d\n",retval);
  return retval;
}

#pragma export(tran1)
#pragma export(tran3)
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;
}

#include <stdio.h>
#include <ctest.h>
#include <leawi.h>

int tran2(int parm1, int parm2, int parm3, long double parm4, int parm5) {
    _INT4 code, timing;
    code = 1001; /* Abend code to issue */
    timing = 1;
    printf("Tran3: About to ABEND\n");
    CEE3ABD(&code,&timing);
    return parm1 + parm2 + parm3;
}

```

**Figure 66: Sample XPLINK-compiled program (tranmain) which calls a NOXPLINK-compiled program**

**Figure 67: Sample NOXPLINK-compiled program (trandll) which calls an XPLINK-compiled program**

---

**Example dump of calling between XPLINK and non-XPLINK programs**

This article displays an example dump of calling between XPLINK and non-XPLINK programs.
Condition Information for Active Routines

Condition Information for CEL4ABD0 (DSA address 2110CA60)
CIB Address: 2110D428
Current Condition: CEE0198S The termination of a thread was signaled due to an unhandled condition.
Original Condition: CEE3260C The system or user abend U0001 R=00000000 was issued.
Location: Program Unit: CEL4ABD0 Entry: CEL4ABD0 Statement: Offset: +0000024C
Machine State:
CPU State:
PSW..... 078D1400 A0AFCC54
GPR0..... 00000000_84000000  GPR1..... 00000000_840003E9  GPR2..... 00000000_00000000  GPR3..... 00000000_2110C9B0

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

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>param5</td>
<td>1431655765</td>
</tr>
<tr>
<td>param4</td>
<td>1.234567889999999977135303197E+03</td>
</tr>
<tr>
<td>param3</td>
<td>57262306</td>
</tr>
<tr>
<td>param2</td>
<td>20863153</td>
</tr>
</tbody>
</table>

Saved Registers:

<table>
<thead>
<tr>
<th>GPR0</th>
<th>20914D70</th>
<th>GPR1</th>
<th>2126C69A8</th>
<th>GPR2</th>
<th>212BBB8F4</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR3</td>
<td>A09B6BDF6</td>
<td>GPR4</td>
<td>21B28B868</td>
<td>GPR5</td>
<td>21B28B8C0</td>
</tr>
<tr>
<td>GPR6</td>
<td>20914D70</td>
<td>GPR7</td>
<td>2126C69A0</td>
<td>GPR8</td>
<td>2090B0C48</td>
</tr>
<tr>
<td>GPR9</td>
<td>20914D70</td>
<td>GPR10</td>
<td>2126C69A0</td>
<td>GPR11</td>
<td>2090B0C48</td>
</tr>
<tr>
<td>GPR12</td>
<td>20914D70</td>
<td>GPR13</td>
<td>2126C69A0</td>
<td>GPR14</td>
<td>2090B0C48</td>
</tr>
<tr>
<td>GPR15</td>
<td>20914D70</td>
<td>GPR16</td>
<td>2126C69A0</td>
<td>GPR17</td>
<td>2090B0C48</td>
</tr>
</tbody>
</table>

GPREG STORAGE:

Storage around GPR0 (20914D70):
-0020 20914D50 00000000 00000000 00000000 00000000 00000000 00000000 |................................|
+0000 20914D70 180F58FF 001007FF 212BB3A0 20914D70 212BB3B0 00000000 00000000 00000000 |.............j(.................|
+0020 20914D90 180F58FF 001007FF 212BB0B0 20914D70 212BB3B0 00000000 00000000 00000000 |.............j(.................|

Local Variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>timing</td>
<td>signed long int 1</td>
</tr>
<tr>
<td>code</td>
<td>signed long int 1061</td>
</tr>
</tbody>
</table>

Condition Information for CEL4ABD0 (DSA address 2110C750)
CIB Address: 211205F8
Current Condition: CEE0198S The termination of a thread was signaled due to an unhandled condition.
Original Condition: CEE3260C The system or user abend U0001 R=00000000 was issued.
Location: Program Unit: CEL4ABD0 Entry: CEL4ABD0 Statement: Offset: +0000024C
Machine State:
CPU State:
PSW..... 078D1400 A0AFCC54
GPR0..... 00000000_84000000  GPR1..... 00000000_840003E9  GPR2..... 00000000_00000000  GPR3..... 00000000_2110C9B0

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

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>param5</td>
<td>1431655765</td>
</tr>
<tr>
<td>param4</td>
<td>1.234567889999999977135303197E+03</td>
</tr>
<tr>
<td>param3</td>
<td>57262306</td>
</tr>
<tr>
<td>param2</td>
<td>20863153</td>
</tr>
</tbody>
</table>

Saved Registers:

<table>
<thead>
<tr>
<th>GPR0</th>
<th>20914D70</th>
<th>GPR1</th>
<th>2126C69A8</th>
<th>GPR2</th>
<th>212BBB8F4</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR3</td>
<td>A09B6BDF6</td>
<td>GPR4</td>
<td>21B28B868</td>
<td>GPR5</td>
<td>21B28B8C0</td>
</tr>
<tr>
<td>GPR6</td>
<td>20914D70</td>
<td>GPR7</td>
<td>2126C69A0</td>
<td>GPR8</td>
<td>2090B0C48</td>
</tr>
<tr>
<td>GPR9</td>
<td>20914D70</td>
<td>GPR10</td>
<td>2126C69A0</td>
<td>GPR11</td>
<td>2090B0C48</td>
</tr>
<tr>
<td>GPR12</td>
<td>20914D70</td>
<td>GPR13</td>
<td>2126C69A0</td>
<td>GPR14</td>
<td>2090B0C48</td>
</tr>
<tr>
<td>GPR15</td>
<td>20914D70</td>
<td>GPR16</td>
<td>2126C69A0</td>
<td>GPR17</td>
<td>2090B0C48</td>
</tr>
</tbody>
</table>

GPREG STORAGE:

Storage around GPR0 (20914D70):
-0020 20914D50 00000000 00000000 00000000 00000000 00000000 00000000 |................................|
+0000 20914D70 180F58FF 001007FF 212BB3A0 20914D70 212BB3B0 00000000 00000000 00000000 |.............j(.................|
+0020 20914D90 180F58FF 001007FF 212BB0B0 20914D70 212BB3B0 00000000 00000000 00000000 |.............j(.................|

Local Variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>timing</td>
<td>signed long int 1</td>
</tr>
<tr>
<td>code</td>
<td>signed long int 1061</td>
</tr>
</tbody>
</table>

Condition Information for CEL4ABD0 (DSA address 2110C750)
### Finding XPLINK information in a Language Environment dump

Table 39 on page 195 describes the specific XPLINK information in sections of the Language Environment dump.
Table 39: Contents of XPLINK information in a Language Environment dump

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Traceback</td>
<td>When an XPLINK-compiled routine calls a NOXPLINK-compiled routine, a glue routine gets control to convert the linkage conventions of the XPLINK caller to those of the NOXPLINK callee. In the sample dump, this routine is CEEVRONU and it appears between main() and tran1() and again between tran2() and tran3(). When a NOXPLINK-compiled routine calls an XPLINK-compiled routine, a glue routine gets control to convert the linkage conventions of the NOXPLINK caller to those of the XPLINK callee. In the sample dump, this routine is CEEVROND and it appears between EDCZHINV and main() and again between tran1() and tran2().</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[2] Parameters, Registers, and Variables for Active Routines</th>
<th>In this section, each DSA is identified as one of the following:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UPSTACK DSA</td>
</tr>
<tr>
<td></td>
<td>The DSA format is that for a NOXPLINK-compiled program that uses an upward growing stack.</td>
</tr>
<tr>
<td></td>
<td>DOWNSTACK DSA</td>
</tr>
<tr>
<td></td>
<td>The DSA format is that for an XPLINK-compiled program that uses a downward growing stack.</td>
</tr>
<tr>
<td></td>
<td>TRANSITION DSA</td>
</tr>
<tr>
<td></td>
<td>The DSA format is that of its callee. A transition DSA can occur between an UPSTACK DSA and a DOWNSTACK DSA where it represents a transition from one linkage convention to another. A transition DSA can also occur between two DOWNSTACK DSAs where it represents a transition from one stack segment to another (a stack overflow).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[3] Control Blocks for Active Routines</th>
<th>In this section, DSAs are formatted. Those previously identified as UPSTACK DSAs will have one format and those identified as DOWNSTACK DSAs will have a different format. Those identified as TRANSITION DSAs will have two parts; the first will be either the downstack or upstack format, the second is unique to transition DSAs and contains information about the transition.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>It is important to understand that the registers saved in an upstack DSA are those saved by a routine that the DSA-owning routine called. Typically register 15 is the entry point of the routine that was called, and register 14 is the return address into the DSA-owning routine. In contrast, the registers saved in an downstack DSA are those saved by the DSA-owning routine on entry. Register 7 is the return address back to the caller of the DSA-owning routine. Register 6 may be the entry point of the DSA-owning routine. (This is not true when the Branch Relative and Save instruction is used to implement the call.)</td>
</tr>
</tbody>
</table>

C/C++ contents of the Language Environment trace tables

Language Environment provides the following C/C++ trace table entry types that contain character data. For more information about the Language Environment trace table format, see "Understanding the trace table entry (TTE)" on page 152.

- 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:

\[
\text{NameOfCallingFunction} \\
\rightarrow (xxx) \text{NameOfCalledFunction}
\]

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

\[
\text{NameOfCallingFunction} \\
\rightarrow (xxx) \text{NameOfCalledFunction} \langle\text{input_parameters}\rangle
\]

In addition, when the call is due to one of these C++ operators:

- \text{-new},
- \text{-new[]},
- \text{-delete},
- \text{-delete[]}

then, the C++ operator will appear and the format becomes:

\[
\text{NameOfCallingFunction} \\
\rightarrow (xxx) \text{NameOfCalledFunction} \langle\text{input_parameters}\rangle \\
\text{NameOfC++Operator}
\]

The format for trace table entry 2 is:

\[
\leftarrow (xxx) \text{R15=\text{value}} \text{ ERRNO=\text{value}}
\]

The format for trace table entry 3 is:

\[
\text{NameOfCallingFunction} \\
\rightarrow (xxx) \text{NameOfCalledFunction}
\]

The format for trace table entry 4 is:

\[
\leftarrow (xxx) \text{R1=\text{xxxxxxxx}} \text{ R2=\text{xxxxxxxx}} \text{ R3=\text{xxxxxxxx}} \text{ ERRNO=\text{xxxxxxxx}} \text{ ERRNO2=\text{xxxxxxxx}}
\]

The format for trace table entry 5, which is shown below, is just like trace table entry 1. The \text{input_parameters} and \text{NameOfC++Operator} only appear for the appropriate functions. The angle brackets (\langle\rangle) indicate that this information does not always appear.

\[
\text{NameOfCallingFunction} \\
\rightarrow (xxx) \text{NameOfCalledFunction} \langle\text{input_parameters}\rangle
\]

The format for trace table entry 6 is:

\[
\leftarrow (xxx) \text{R1=\text{xxxxxxxx}} \text{ R2=\text{xxxxxxxx}} \text{ R3=\text{xxxxxxxx}} \text{ ERRNO=\text{xxxxxxxx}} \text{ ERRNO2=\text{xxxxxxxx}}
\]

In all entry types, (xxx) and (xxxx) are numbers associated with the called library function and are used to associate a specific entry record with its corresponding return record.

For entry types 5 and 6, the number will be the same as the number of the function as seen in the C runtime library definition side-deck, SCEELIB dataset member CELHS003, on the IMPORT statement for that function.
The format for trace table entry 7 is:

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

The format for trace table entry 8 is:

```
ModuleNameOfCallingFunction:NameOfCallingNonXplinkFunction
-->ModuleNameOfCalledFunction:NameOfCalledXplinkFunction
```

For entry types 7 and 8, 16 bytes is for the module name and 32 bytes is for the function name. If the name is longer than 16 or 32 bytes, an extra trace entry is taken. The name is truncated and only the first 32/64(16/32) bytes will appear in the trace table entry. Also, a module name might not always be located, such as when a DLL is freed. If that occurs, "UNKNOWN" appears for the module name in the trace table entry.

The below trace table shows a non-XPLINK trace that has examples of C/C++ trace table entry types 1 thru 4.
The code below shows an XPLINK trace that has examples of the trace entries 5 and 6.

```
...The code below shows an XPLINK trace that has examples of the trace entries 5 and 6.
...```

```
```
Figure 68 on page 200 shows an example of the format of the trace table entry type 7 and 8.

<table>
<thead>
<tr>
<th>Time 22.41.35.439976</th>
<th>Date 2001.08.30</th>
<th>Thread ID... 26C70D0000000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member ID.... 03</td>
<td>Flags...... 000000</td>
<td>Entry Type..... 00000006</td>
</tr>
</tbody>
</table>

The following is an example of a dump of the trace table when you specify the LE=20 suboption.

### Language Environment Trace Table:

The most recent trace entry is at displacement: 000000

<table>
<thead>
<tr>
<th>Displacement</th>
<th>Trace Entry in Hexadecimal</th>
<th>Trace Entry in EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>Time 22.10.56.804695</td>
<td>Date 2005.05.01</td>
</tr>
<tr>
<td>Member ID...</td>
<td>Flags...... 000000</td>
<td>Entry Type.... 00000006</td>
</tr>
<tr>
<td>000010</td>
<td>Time 22.10.56.799195</td>
<td>Date 2005.05.01</td>
</tr>
<tr>
<td>Member ID...</td>
<td>Flags...... 000000</td>
<td>Entry Type.... 00000008</td>
</tr>
<tr>
<td>000018</td>
<td>Time 22.10.56.804695</td>
<td>Date 2005.05.01</td>
</tr>
<tr>
<td>Member ID...</td>
<td>Flags...... 000000</td>
<td>Entry Type.... 00000007</td>
</tr>
<tr>
<td>000028</td>
<td>Time 22.10.56.804695</td>
<td>Date 2005.05.01</td>
</tr>
<tr>
<td>Member ID...</td>
<td>Flags...... 000000</td>
<td>Entry Type.... 00000009</td>
</tr>
<tr>
<td>000036</td>
<td>Time 22.10.56.804695</td>
<td>Date 2005.05.01</td>
</tr>
<tr>
<td>Member ID...</td>
<td>Flags...... 000000</td>
<td>Entry Type.... 0000000A</td>
</tr>
</tbody>
</table>

---

**Figure 68: Trace table with trace table entry types 7 and 8**
Debugging C/C++ routines
Additional Language Specific Information:

errno information:
Thread Id .... 21DEC8300000000 Erno ...... 0 Ernojr .... 00000000
Debugging examples of C/C++ routines

This section contains examples that demonstrate the debugging process for C/C++ routines. Important areas of the output are highlighted. Data unnecessary to the debugging examples has been replaced by ellipses.

Divide-by-zero error

Figure 69 on page 203 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 69: C routine with a divide-by-zero error*

To debug this routine, use the following steps:

1. Locate the Original Condition message in the Condition Information for Active Routines section of the dump. In this example, the message is CEE3209S. The system detected a fixed—point divide exception. This message indicates the error was caused by an attempt to divide by zero. For more information about CEE3209S, see *z/OS Language Environment Runtime Messages*.

The traceback section of the dump indicates that the exception occurred at offset X'76' within function funcb. This information is used along with the compiler-generated Pseudo Assembly Listing to determine where the problem occurred.

If the TEST compiler option is specified, variable information is in the dump. If the GONUMBER compiler option is specified, statement number information is in the dump. *Figure 70 on page 204* shows the generated traceback from the dump.
2. Locate the instruction with the divide-by-zero error in the Pseudo Assembly Listing in Figure 71 on page 205.

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 172.

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 72 on page 206 shows the WSA address.

---

**Figure 71: Pseudo assembly listing (C/C++ routine divide-by-zero error)**

---

```
<table>
<thead>
<tr>
<th>OFFSET</th>
<th>OBJECT CODE</th>
<th>LINE#</th>
<th>FILE#</th>
<th>PSEUDO ASSEMBLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>CEEA106</td>
<td></td>
<td></td>
<td>Flags</td>
</tr>
<tr>
<td>000008</td>
<td>0800022B</td>
<td></td>
<td></td>
<td>@PPA3-main</td>
</tr>
<tr>
<td>000008</td>
<td>0800017B</td>
<td></td>
<td></td>
<td>@PPA3-main</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>No EPD</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>Register save mask</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>Member flags</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>Flags</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>Flags</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>Member flags</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>Offset/2 to COL</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>Offset/2 to COL</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>CDC function length/2</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>CDC function EP offset</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>CDC prolog</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>CDC end</td>
</tr>
<tr>
<td>000008</td>
<td>08000000</td>
<td></td>
<td></td>
<td>AL2(4),’main’</td>
</tr>
<tr>
<td>000000</td>
<td>D985A2A4</td>
<td>93A34070</td>
<td>40A3841</td>
<td></td>
</tr>
</tbody>
</table>

Constant Area

<table>
<thead>
<tr>
<th>Result = Rd..</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000080</td>
<td>Constant Area</td>
</tr>
</tbody>
</table>

```
Enclave Control
Blocks:

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

Figure 72: C/C++ CAA information in dump (C/C++ routine divide-by-zero error)

4. Routines compiled with the RENT option must also be processed by the binder. The binder produces the Writable Static Map. Find the offset of `statint` in the Writable Static Map in Figure 73 on page 206. In this example, the offset is X'0'.

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OFFSET</th>
<th>NAME</th>
<th>TYPE</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_WSA</td>
<td>0</td>
<td>statint</td>
<td>PART</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>fa</td>
<td>PART</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>environ</td>
<td>PART</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>errno</td>
<td>PART</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 73: Writable static map (C/C++ routine divide-by-zero error)

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++ runtime for the C/C++ user, so it is in the user heap. For a program object, the writable static area is storage allocated by the loader and is shown in the WSA for Program Object(s) section of the dump.

For this example, the program was built as a program object. The writable static area is displayed in the Enclave Storage section of the dump, shown in Figure 74 on page 207.

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 75 on page 207 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.

```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 75: 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 76 on page 208. In this example, the message is CEE3201S The system detected an operation exception (System Completion Code=0C1). This message suggests that the error was caused by an attempt to branch to an unknown address. For additional information about CEE3201S, see z/OS Language Environment Runtime Messages.

The Location section of the dump indicates that the exception occurred at offset X’-20900978’ within function funca and that there may have been a bad branch from offset X’0000005A’ within function funca. The negative offset indicates that the offset cannot be used to locate the instruction that caused the error. Another indication of bad data is the value of X’80000002’ in the instruction address of the PSW. This address indicates that an instruction in the routine branched outside the bounds of the routine.
Condition Information for Active Routines

Condition Information for (DSA address 20FCB2B0)

CEEBR998S The termination of a thread was signaled due to an unhandled condition.

Original Condition:

CEEBR915S The system detected an operation exception (System Completion Code=0C1).

Location:

Program Unit: Entry: funca Statement: Offset: -20900978
Possible Bad Branch: Statement: Offset: +0000005A

Machine State:

ILC..... 0002 Interruption Code..... 0001
PSW..... 078D1400 80000002
GPR0..... 00000000_20FCB350  GPR1..... 00000000_20FCB2A0  GPR2..... 00000000_20FCB2A0  GPR3..... 00000000_209009B2
GPR4..... 00000000_A09A0BBC  GPR5..... 00000000_20912648  GPR6..... 00000000_20900AA4  GPR7..... 00000000_20900098
GPR8..... 00000000_00000030  GPR9..... 00000000_80000000  GPR10.... 00000000_A0E699E2  GPR11.... 00000000_A09A0AD8
GPR12.... 00000000_209139B0  GPR13.... 00000000_20FCB2B0  GPR14.... 00000000_A09009D4  GPR15.... 00000000_00000000

Storage dump near condition, beginning at location: 00000000

GPREG STORAGE:

Storage around GPR0 (20FCB350)

-0020 20FCB330  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  |................................|
+0000 20FCB350  0808CEE1 20FCB2B0 20FCE470 A09D6B3A  A09EFFD8 20FCB350 20FCB7A8 20912648  |..........U...,....Q...&...y.j..|
⋮

Parameters, Registers, and Variables for Active Routines:

funca (DSA address 20FCB2B0):

Saved Registers:

GPR0..... 20FCB350  GPR1..... 20FCB2A0  GPR2..... 20FCB2A0  GPR3..... 209009B2
GPR4..... A09A0BBC  GPR5..... 20912648  GPR6..... 20900AA4  GPR7..... 20900098
GPR8..... 00000030  GPR9..... 80000000  GPR10.... A0E699E2  GPR11.... A09A0AD8
GPR12.... 209139B0  GPR13.... 20FCB2B0  GPR14.... A09009D4  GPR15.... 00000000
⋮

Figure 76: Sections of the dump from example C routine (calling a nonexistent subroutine)
2. Find the branch instructions at offset X'+0000005A' of funca in the listing in Figure 77 on page 209. The instruction is BASR r14,r15. This branch is part of the source statement *aa = func_ptr().

Figure 77: Pseudo assembly listing (calling a nonexistent subroutine)
3. Find the offset of $\text{func\_ptr}$ in the Writable Static Map, shown in Figure 78 on page 210, as produced by the binder.

```
CLASS  C_WSA             LENGTH =       44  ATTRIBUTES = MRG, DEFER ,
              RMODE=ANY
              OFFSET =        0 IN SEGMENT 002       ALIGN =

CLASS
SECTION  OFFSET  NAME                TYPE    LENGTH
func_ptr  0       func_ptr         PART             4
environ   8       environ          PART             4
erro      10       errno            PART             4
tzname    18       tzname           PART             8
```

Figure 78: Writable static map (calling a nonexistent subroutine)

4. Add the offset of $\text{FUNC\_PTR}(\text{X'0'})$ to the address of WSA (X'20914F58'). The result (X'20914F58') is the address of the function pointer $\text{func\_ptr}$ in the writable static storage area within the heap. This value is 0, indicating the variable is uninitialized. Figure 79 on page 210 shows the sections of the dump.

```
Enclave Control Blocks:

Enclave Storage:

Figure 79: Enclave control blocks and storage sections in dump (calling a nonexistent subroutine)
```

**Calling a nonexistent XPLINK function**

Figure 80 on page 211 demonstrates the error of calling a nonexistent function. This routine was compiled with the compiler options XPLINK, LIST and RENT and was run with the option TERMTHDACT(DUMP). This routine was not compiled with the TEST(ALL) compile option. As a result, arguments and variables do not appear in the dump.
To debug this routine, use the following steps:

1. Locate the Original Condition message in the Condition Information for Active Routines section of the dump, shown in “Sections of the dump from example C routine (calling a nonexistent XPLINK function)” on page 212. In this example, the message is CEE3201S The system detected an operation exception (System Completion Code=0C1). This message suggests that the error was caused by an attempt to branch to an unknown address. For additional information about CEE3201S, see z/OS Language Environment Runtime Messages.

   The location section of the dump indicates that the exception occurred at offset X’-20900158’ within function funca and that there may have been a bad branch from offset X’+0000001C’. The negative offset indicates that the offset cannot be used to locate the instruction that caused the error. Another indication of bad data is the value of X’80000004’ in the instruction address of the PSW. This address indicates that an instruction in the routine branched outside the bounds of the routine.

2. Find the branch instruction at offset X’+0000001C’ of funca in the listing in Figure 81 on page 211. This instruction is BASR r7,r6. This branch is part of the source statement *aa = func_ptr().

Figure 80: C/C++ example of calling a nonexistent XPLINK function

Figure 81: Pseudo assembly listing (calling a nonexistent XPLINK function)
3. Find the offset of `func_ptr` in theWritable Static Map, shown in Figure 82 on page 212.

Figure 82: Writable static map (calling a nonexistent XPLINK function)

4. Add the offset of `func_ptr` (X'38') to the address of WSA (X'20914FC0'). The result (X'20914FF8') is the address of the function pointer `func_ptr` in the writable static storage area within the heap. This value is 0, indicating the variable is uninitialized. Figure 83 on page 212 shows the sections of the dump.

Figure 83: Enclave control blocks and storage sections in dump (calling a nonexistent XPLINK function)

Sections of the dump from example C routine (calling a nonexistent XPLINK function)
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 spawnp(), vfork(), or one of the exec family of functions, the SYSMDUMP DD allocation information is not inherited. Even though the SYSMDUMP allocation is not inherited, a SYSMDUMP allocation must exist in the parent in order to obtain a HFS storage dump.

Alternatively, you can specify the DYNDUMP runtime option to generate a system dump. For more information, see z/OS Language Environment Programming Reference.

If the program terminates abnormally while running in this new address space, the kernel causes an unformatted storage dump to be written to an HFS file in the user's working directory. The file is placed in the current working directory or into /tmp if the current working directory is not defined. The file name has the following format; directory is the current working directory or tmp, and pid is the hexadecimal process ID (PID) for the process that terminated. For details on how to generate the system dump, see “Steps for generating a system dump in a z/OS UNIX shell” on page 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:
Figure 84: 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; details on how to request and interpret the reports are provided in the following sections.

- By setting the Language Environment RPTSTG(ON) runtime option for Language Environment created heaps
- By issuing a stand-alone call to the C function __uheapreport() for user–created heaps.

### Language Environment storage report with heap pools statistics

To request a Language Environment storage report set RPTSTG(ON). If the C/C++ application specified the HEAPPOOLS(ON) runtime option, then the storage report displays heap pools statistics. Figure 4 on page 14 is a sample storage report that shows heap pools statistics for a multithreaded C/C++ application. The following sections describe the C/C++ specific heap pools information.
HEAPPOOLS storage statistics
The HEAPPOOLS runtime option controls usage of the heap pools storage algorithm at the enclave level. The heap pools algorithm allows for the definition of one to twelve heap pools, each consisting of a number of storage cells of a specified length.

Note: The use of an alternative Vendor Heap Manager (VHM) overrides the use of the HEAPPOOLS runtime option.

HEAPPOOLS statistics
• Pool p size: ssss Get requests: gggg
  p the number of the pool. When there are multiple pools for a cell size, the pools are numbered using the format aa.bbb.
  aa the number for the cell size.
  bbb the number for the pool within the cell size.
  ssss the cell size specified for the pool.
  gggg the number of storage requests that were satisfied from this pool.
• Successful Get Heap requests: xxxx-yyyy n
  xxxx the low side of the 8 byte range
  yyyy the high side of the 8 byte range
  n the number of requests in the 8 byte range.
• Requests greater than the largest cell size — the number of storage requests that are not satisfied by heap pools.
Note: Values displayed in the HEAPPOOLS statistics report are not serialized when collected; therefore, the values are not necessarily exact.

HEAPPOOLS summary
The HEAPPOOLS summary displays a report of the HEAPPOOLS statistics and provides suggested percentages for current cell sizes as well as suggested cell sizes.
• Specified Cell Size — the size of the cell specified in the HEAPPOOLS runtime option
• Element Size — the size of the cell plus any additional storage needed for control information or to maintain alignment
• Extent Percent — the cell pool percent specified by the HEAPPOOLS runtime option
• Cells Per Extent — the number of cells per extent. This number is calculated using the following formula, with a minimum of four cells:

\[
\text{Initial Heap Size} \times \left(\frac{\text{Extent Percent}}{100}\right) / \text{(Element Size)}
\]

Note: Having a small number of cells per extent is not recommended since the pool could allocate many extents, which would cause the HEAPPOOLS algorithm to perform inefficiently.
• Extents Allocated — the number of times that each pool allocated an extent.
To optimize storage usage, the extents allocated should be either one or two. If the number of extents allocated is too high, then increase the percentage for the pool.
• Maximum Cells Used — the maximum number of cells used for each pool.

• Cells In Use — the number of cells that were never freed.

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

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

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

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

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

• Suggested Cell Sizes — sizes that are calculated to optimally use storage (assuming that the application will __malloc__/__free with the same frequency).

The suggested cell sizes are given with no percentages because the usage of each new cell pool size is not known. If there are less than 12 cell sizes calculated and the last calculated cell size is smaller than the largest cell size currently in effect, the largest cell size currently in effect will be used for the last suggested cell size.

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

**C function __uheapreport() storage report**

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

Figure 85 on page 216 shows a sample storage report generated by __uheapreport(). For more information on the __uheapreport() function, see z/OS XL C/C++ Runtime Library Reference. For tuning tips, see z/OS Language Environment Programming Guide.

---

Storage Report for Enclave  12/26/09 11:42:23 AM
Language Environment V01 R12.00

HeapPools Statistics:
Pool 1 size:    32
Successful Get Heap requests:     1-   32          11250
Pool 2 size:   128
Successful Get Heap requests:    97-  128           3396
Pool 3 size:   512
Successful Get Heap requests:   481-  512            864
Pool 4 size:  2048
Successful Get Heap requests:  2017- 2048            216
Pool 5 size:  8192
Successful Get Heap requests:  8161- 8192             54
Pool 6 size: 16384
Successful Get Heap requests: 16353-16384             27
Requests greater than the largest cell size:             0

HeapPools Summary:

<table>
<thead>
<tr>
<th>Cell Size</th>
<th>Extent</th>
<th>Cells Per Extent</th>
<th>Extents Allocated</th>
<th>Maximum Cells Used</th>
<th>Cells In Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>15</td>
<td>3750</td>
<td>1</td>
<td>3750</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>15</td>
<td>1102</td>
<td>1</td>
<td>1102</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>15</td>
<td>288</td>
<td>1</td>
<td>288</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>15</td>
<td>72</td>
<td>1</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>15</td>
<td>18</td>
<td>1</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>15</td>
<td>9</td>
<td>1</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

Suggested Percentages for current Cell Sizes:
32,15,128,15,512,15,2048,15,8192,15,16384,15

Suggested Cell Sizes:
32,128,512,2048,8192,16384,

End of Storage Report

Figure 85: Storage report generated by __uheapreport()
User-created HeapPools statistics

- Pool p size: ssss
  
  \( p \)
  the number of the pool
  
  \( ssss \)
  the cell size specified for the pool.

- Successful Get Heap requests: xxxx-yyyy n
  
  \( xxxx \)
  the low side of the range
  
  \( yyyy \)
  the high side of the range
  
  \( n \)
  the number of requests in the range.

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

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

HeapPools summary

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

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

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

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 __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.
MEMCHECK VHM memory leak analysis tool

The MEMCHECK VHM memory leak analysis tool is an alternative vendor heap manager used to diagnose memory problems. MEMCHECK VHM performs the following functions and displays the results in two reports:

- Check for heap storage leaks, double free, and overlays.
- Trace user heap storage allocation and deallocation requests.

The trace is limited to 1024 entries and will wrap.

Restrictions:

- MEMCHECK VHM works with C/C++ and Enterprise PL/I applications, but is not enabled for COBOL or Fortran.
- MEMCHECK VHM and HEAPPOOLS are mutually exclusive. HEAPPOOLS will be ignored when MEMCHECK VHM is active.
- MEMCHECK VHM should not be used in PIPI, PICI, CICS, and SPC environments.

Invoking MEMCHECK VHM

As with any alternate vendor heap manager, you must specify the dllname with the environment variable _CEE_HEAP_MANAGER to indicate that MEMCHECK VHM will be used to manage the user heap. Since _CEE_HEAP_MANAGER must be set before any user code gains control, use the ENVAR runtime option to set the variable or set it inside the file specified by environment variables _CEE_ENVFILE or _CEE_ENVFILE_S. The format follows:

```plaintext
_CEEL_HEAP_MANAGER=dllname
```

The following two DLLs are associated with MEMCHECK VHM and use the following events.

- CEL4MCHK: 31-bit base and XPLINK
- CELQMCHK: 64-bit

_VHM_INIT

replaces C-RTL malloc(), calloc(), realloc(), and free() with the corresponding MEMCHECK VHM functions. This event is only at Language Environment Initialization and only called by Language Environment.

_VHM_TERM

terminates Vendor Heap Manager to free the memcheck storage functions. This event is called only by Language Environment at Language Environment Termination.

_VHM_REPORT

generates the Heap Leak Report and the optional Trace Report. This new event will be called by Language Environment at Language Environment Termination and will write the Heap Leak Report (and the optional Trace Report if the _CEE_MEMCHECK_TRACE environment variable is active) in the output file name specified in _CEE_MEMCHECK_OUTFILENAME. This event can also be called dynamically by the __vhm_event() API.

MEMCHECK VHM environment variables

The MEMCHECK VHM environment variables control the tool, the call levels of the Heap Leak Report and Trace Report, the Overlay Analysis, the pad length added in the user heap allocation for overlay analysis, and the output file name for the reports. They should be activated through the ENVAR runtime option, the file specified by the _CEE_ENVFILE (or _CEE_ENVFILE_S) environment variable, or using the export command from the z/OS UNIX shell before any user code gets control (prior to the HLL user exit, static
constructors, or main getting control). Setting these environment variables after the user code has begun execution will not activate them and the default values will be used.

**_CEE_MEMCHECK_DEPTH_**

*Description:* Controls the number of call-levels to be generated on the Heap Leak Report.

*Valid settings:* integer value: the minimum is 1 and the maximum is 100. If the value specified is not valid, the default will be used.

*Default:* 10.

**_CEE_MEMCHECK_OVERLAY_**

*Description:* Activates the storage overlays analysis beyond the end of the malloc'd storage.

*Valid settings:* ON to activate the analysis, OFF to deactivate. If an invalid value is specified, the default value will be used.

*Default:* OFF

**_CEE_MEMCHECK_OVERLAYLEN_**

*Description:* Sets the pad length added in the user heap allocation for overlay analysis. This environment variable will be used only if _CEE_MEMCHECK_OVERLAY is active.

*Valid settings:* integer value, multiple of 8: the minimum is 8 and the maximum is 80. Non-multiples of 8 will be rounded up to the next multiple.

*Default:* 8

**_CEE_MEMCHECK_TRACE_**

*Description:* Enables tracing of all heap storage allocation and deallocation and a Trace Report will be generated at Language Environment Termination.

*Valid settings:* ON to activate the analysis, OFF to deactivate. If an invalid value is specified, the default value will be used.

*Default:* OFF

**_CEE_MEMTRACE_DEPTH_**

*Description:* Controls the number of call-levels to be generated in the Trace Report, on each call to a library function that deals with heap. This environment variable will be used only if _CEE_MEMCHECK_TRACE is active.

*Valid settings:* integer value: the minimum is 1 and the maximum is 100. If the value specified is not valid, the default value will be used.

*Default:* 10

**_CEE_MEMCHECK_OUTFILENAME_**

*Description:* Sets the name of the fully qualified path name of the file in which the Heap Leak Report and Trace Report should be directed. The report name could be any valid name used in C-RTL fopen() function, then it could also generates the reports in a Data Set.

*Valid settings:* string value. If an invalid value is specified, the default value will be used.

*Default:* standard error output

**MEMCHECK VHM report sample scenario**

In this example, the MEMCHECK VHM tool is used by specifying the environment variables from the z/OS UNIX shell. The user specifies a depth of 8 call levels in the Heap Leak Report and 8 call levels in the Trace Report for 31-bit.

1. Specifies the depth to trace on storage requests (written to the Heap Leak Report):

   ```bash
   Export _CEE_MEMCHECK_DEPTH=8
   ```
2. Activates the Trace Report option:

   Export _CEE_MEMCHECK_TRACE=ON

3. Specifies the depth to trace on storage requests (written to the Trace Report):

   Export _CEE_MEMTRACE_DEPTH=8

4. Activates the Overlay analysis option:

   Export _CEE_MEMCHECK_OVERLAY=ON

5. Activates the tool with the 31-bit DLL (automatically generating the Heap Leak Report):

   Export _CEE_HEAP_MANAGER=CEL4MCHK

**MEMCHECK VHM report examples**

Both reports are written at Language Environment termination (_VHM_TERM event). They are written in the output file name specified in _CEE_MEMCHECK_OUTFILENAME and are consistent with the format of other Language Environment reports.

The following trace report will be generated at Language Environment termination (_VHM_TERM event) if the _CEE_MEMCHECK_TRACE environment variable is active. The report generates the traceback information of all heap storage allocations and deallocations.
The Heap Leak Report (Figure 86 on page 222) 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: 25601b68 +0000097e function2
  Called from: 25601c68 +00000ca function1
  Called from: 25601a60 +00000062 main

Unmatched ALLOCATE of 48 bytes at address 0x25a2ec90
- sequence 4
  Called from: 25a44330 +000000d2 MemAlloc
  Called from: 05cd9918 +0000005c CEEPGTFN
  Called from: 25725c08 +000000a0 dllinit
  Called from: 05d49c88 +000007dc (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: 25862540 +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: 25601c68 +000000bc function1
Called from: 25601a60 +00000062 main

Figure 86: Heap Leak Report generated by MEMCHECK VHM

The following names are used within MEMCHECK to denote special cases and may be displayed in any of
the reports:

(unknown)
Name of the routine is not known.

(nome)
Routine does not have a name in the PPA section. (For example, module compiled with compress
option).

(nospace)
Internal memory space reserved by MEMCHECK is full, so name was not saved for the traceback
information. No action is needed from the user.
Chapter 5. Debugging COBOL programs

This section provides information for debugging applications that contain one or more COBOL programs. It includes information about:

• Determining the source of error
• Generating COBOL listings and the Language Environment dump
• Finding COBOL information in a dump
• Debugging example COBOL programs

Determining the source of error

The following sections describe how you can determine the source of error in your COBOL program. They explain how to simplify the process of debugging COBOL programs by using features such as the DISPLAY statement, declaratives, and file status keys. The following methods for determining errors are covered:

• Tracing program logic
• Finding and handling input/output errors
• Validating data
• Assessing switch problems
• Generating information about procedures

After you have located and fixed any problems in your program, you should delete all debugging aids and recompile it before running it in production. Doing so helps the program run more efficiently and use less storage.

For detailed information about any of the topics and techniques discussed in the following sections, refer to the appropriate COBOL documentation in the Enterprise COBOL for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036733).

Tracing program logic

You can add DISPLAY statements to help you trace through the logic of the program in a non-CICS environment. If, for example, you determine that the problem appears in an EVALUATE statement or in a set of nested IF statements, DISPLAY statements in each path tell you how the logic flows. You can also use DISPLAY statements to show you the value of interim results. Scope terminators can also help you trace the logic of your program because they clearly indicate the end of a statement.

For example, to check logic flow, you might insert the following statement to determine if you started and finished a particular procedure:

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

After you are sure that the program works correctly, comment out the DISPLAY statement lines by putting asterisks in position 7 of the appropriate lines.

Finding input/output errors

VSAM file status keys can help you determine whether routine errors are due to the logic of your routine or are I/O errors occurring on the storage media. To use file status keys as a debugging aid, include a test
after each I/O statement to check for a value other than 0 in the file status key. If the value is other than 0, you can expect to receive an error message. You can use a nonzero value to indicate how the I/O procedures in the routine were coded. You can also include procedures to correct the error based on the file status key value.

Handling input/output errors

If you have determined that the problem lies in one of the I/O procedures in your program, you can include the USE EXCEPTION/ERROR declarative to help debug the problem. If the file does not open, the appropriate USE EXCEPTION/ERROR declarative is activated. You can specify the appropriate declarative for the file or for the different open attributes: INPUT, OUTPUT, I/O, or EXTEND. Code each USE AFTER STANDARD ERROR statement in a separate section immediately after the Declarative Section keyword of the Procedure Division.

Validating data (class test)

If you suspect that your program is trying to perform arithmetic on nonnumeric data or is somehow receiving the wrong type of data on an input record, you can use the class test to validate the type of data.

Assessing switch problems

Using INITIALIZE or SET statements to initialize a table or data item is useful when you suspect that a problem is caused by residual data left in those fields. If your problem occurs intermittently and not always with the same data, the problem could be that a switch is not initialized, but is generally set to the right value (0 or 1). By including a SET statement to ensure that the switch is initialized, you can determine if the uninitialized switch is the cause of the problem.

Generating information about procedures

You can use the USE FOR DEBUGGING declarative to include COBOL statements in a COBOL program and specify when they should run. Use these statements to generate information about your program and how it is running. Code each USE FOR DEBUGGING declarative in a separate section in the DECLARATIVES SECTION of the PROCEDURE DIVISION.

For example, to check how many times a procedure is run, include a special procedure for debugging (in the USE FOR DEBUGGING declarative) that adds 1 to a counter each time control passes to that procedure. The adding-to-a-counter technique can be used as a check for:

• How many times a PERFORM ran. This shows you whether the control flow you are using is correct.
• How many times a loop routine actually runs. This tells you whether the loop is running and whether the number you have used for the loop is accurate.

You can use debugging lines, debugging statements, or both in your program. Debugging lines are placed in your program, and are identified by a D in position 7. Debugging statements are coded in the DECLARATIVES SECTION of the PROCEDURE DIVISION.

• The USE FOR DEBUGGING declaratives must:
  – Be only in the DECLARATIVES SECTION
  – Follow a DECLARATIVES header USE FOR DEBUGGING

  With USE FOR DEBUGGING, the TEST compiler option must have the NONE hook-location suboption specified or the NOTEST compiler option must be specified. The TEST compiler option and the DEBUG runtime option are mutually exclusive, with DEBUG taking precedence.

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

To use debugging lines and statements in your declarative procedures, you must include both:

• WITH DEBUGGING MODE in the SOURCE-COMPUTER paragraph in the ENVIRONMENT DIVISION
• The DEBUG runtime option

224 z/OS: Language Environment Debugging Guide
Figure 87 on page 225 shows how to use the DISPLAY statement and the USE FOR DEBUGGING declarative to debug a program.

Environment Division
Source Computer . . . With Debugging Mode.

Data Division.

File Section.

Working-Storage Section.
*(among other entries you would need:)
01 Trace-Msg PIC X(30) Value "Trace for Procedure-Name : ".
01 Total PIC 99 Value Zeros.
*(balance of Working-Storage Section)

Procedure Division.
Declaratives.
Debug-Declar Section.
Use For Debugging On 501-Some-Routine.
Debug-Declar-Paragraph.
Display Trace-Msg, Debug-Name, Total.
Debug-Declar-End.
Exit.

End Declaratives.
Begin-Program Section.
: Perform 501-Some-Routine.
*(within the module where you want to test, place:)
Add 1 To Total
* (whether you put a period at the end depends on
* where you put this statement.)

Figure 87: Example of using the WITH DEBUGGING MODE clause

In the example in Figure 87 on page 225, portions of a program are shown to illustrate the kind of statements needed to use the USE FOR DEBUGGING declarative. The DISPLAY statement specified in the DECLARATIVES SECTION issues the following message every time the PERFORM 501-SOME-ROUTINE runs. The total shown, nn, is the value accumulated in the data item named TOTAL:

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

Another use for the DISPLAY statement technique shown above is to show the flow through your program. You do this by changing the USE FOR DEBUGGING declarative in the DECLARATIVES SECTION to the following value and dropping the word TOTAL from the DISPLAY statement.

USE FOR DEBUGGING ON ALL PROCEDURES.

Using COBOL listings

When you are debugging, you can use one or more of the listings shown in Table 40 on page 226. The following sections give an overview of each of these listings and the compiler option you use to obtain each listing.
Table 40: Compiler-generated COBOL listings and their contents

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted Cross-Reference</td>
<td>Provides sorted cross-reference listings of DATA DIVISION, PROCEDURE DIVISION, and program names. The listings provide the location of all references to this information.</td>
<td>XREF</td>
</tr>
<tr>
<td>Listings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Map listing</td>
<td>Provides information about the locations of all DATA DIVISION items and all implicitly declared variables. This option also supplies a nested program map, which indicates where the programs are defined and provides program attribute information.</td>
<td>MAP</td>
</tr>
<tr>
<td>Verb Cross-Reference</td>
<td>Produces an alphabetic listing of all the verbs in your program and indicates where each is referenced.</td>
<td>VBREF</td>
</tr>
<tr>
<td>listing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure Division</td>
<td>Tells the COBOL compiler to generate a listing of the PROCEDURE DIVISION along with the assembler coding produced by the compiler. The list output includes the assembler source code, a map of the task global table (TGT), information about the location and size of WORKING-STORAGE and control blocks, and information about the location of literals and code for dynamic storage usage.</td>
<td>LIST</td>
</tr>
<tr>
<td>listings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure Division</td>
<td>Instead of the full PROCEDURE DIVISION listing with assembler expansion information, you can use the OFFSET compiler option to get a condensed listing that provides information about the program verb usage, global tables, WORKING-STORAGE, and literals. The OFFSET option takes precedence over the LIST option. That is, OFFSET and LIST are mutually exclusive; if you specify both, only OFFSET takes effect.</td>
<td>OFFSET</td>
</tr>
<tr>
<td>listings</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Generating a Language Environment dump of a COBOL program

The sample programs shown in Figure 88 on page 226 and Figure 89 on page 227 generate Language Environment dumps with COBOL-specific information.

COBOL program that calls another COBOL program

In Figure 88 on page 226, program COBDUMP1 calls COBDUMP2, which in turn calls the Language Environment dump service CEE3DMP.

```cobol
CBL TEST(STMT,SYM),RENT
IDENTIFICATION DIVISION.
PROGRAM-ID. COBDUMP1.
AUTHOR. USER NAME
ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 SOME-WORKINGSTG.
   05 SUB-LEVEL PIC X(80).
01 SALARY-RECORDA.
   02 NAMEA PIC X(10).
   02 DEPTA PIC 9(4).
   02 SALARYA PIC 9(6).
PROCEDURE DIVISION.
START-SEC.
   DISPLAY 'STARTING TEST COBDUMP1'.
   MOVE 'THIS IS IN WORKING STORAGE' TO SUB-LEVEL.
   CALL 'COBDUMP2' USING SALARY-RECORDA.
   DISPLAY 'END OF TEST COBDUMP1'.
   STOP RUN
END PROGRAM COBDUMP1.
```

Figure 88: COBOL program COBDUMP1 calling COBDUMP2
COBOL program that calls the Language Environment CEE3DMP callable service

In the example in Figure 89 on page 227, program COBDUMP2 calls the Language Environment dump service CEE3DMP.

```cobol
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 IOFSS1R 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.
```

Figure 89: COBOL program COBDUMP2 calling the Language Environment dump service CEE3DMP

Sample Language Environment dump with COBOL-specific information

The call in program COBDUMP2 to CEE3DMP generates a Language Environment dump, shown below. The dump includes a traceback section, which shows the names of both programs, a section on register usage at the time the dump was generated, and a variables section, which shows the storage and data items for each program. Note that the high half of register 14 at entry to CEE3DMP is not available and is shown in the dump as ********. Character fields in the dump are indicated by single quotes. For an explanation of these sections of the dump, see “Finding COBOL information in a dump” on page 228.

```
CEE3DMP V1 R12.0: COBOL DUMP                                                        07/10/10 2:59:23 PM                  Page:    1
ASID: 0099   Job ID: J0005740   Job name: COBDUMP1   Step name: GO         UserID: BARBARA
CEE3845I CEEDUMP Processing started.
CEE3DMP called by program unit COBDUMP2 at statement 40 (offset +00000496).
Registers on Entry to CEE3DMP:
PM....... 0000
GPR0..... 00000000_11480BDC  GPR1..... 00000000_114842C0  GPR2..... 00000000_114A4340  GPR3..... 00000000_11202BBC
GPR4..... 00000000_11202818  GPR5..... 00000000_11480100  GPR6..... 00000000_00000000  GPR7..... 00000000_00FDD100
GPR8...... 00000000_114A41D8  GPR9..... 00000000_11480AA0  GPR10.... 00000000_11202908  GPR11.... 00000000_11202AD4
GPR12.... 00000000_112129C0  GPR13.... 00000000_114841D0  GPR14.... ********_91202C78  GPR15.... 00000000_12EF898
FPR0..... 00000000  00000000            FPR2..... 00000000  00000000
FPR4..... 00000000  00000000            FPR6..... 00000000  00000000
VR0...... 00000000 00000000 00000000 00000000        VR1...... 00000000 00000000 00000000 00000000
VR2...... 00000000 00000000 00000000 00000000        VR3...... 00000000 00000000 00000000 00000000
VR4...... 00000000 00000000 00000000 00000000        VR5...... 00000000 00000000 00000000 00000000
VR6...... 00000000 00000000 00000000 00000000        VR7...... 00000000 00000000 00000000 00000000
VR8...... 00000000 00000000 00000000 00000000        VR9...... 00000000 00000000 00000000 00000000
VR10..... 00000000 00000000 00000000 00000000        VR11..... 00000000 00000000 00000000 00000000
VR12..... 00000000 00000000 00000000 00000000        VR13..... 00000000 00000000 00000000 00000000
VR14..... 00000000 00000000 00000000 00000000        VR15..... 00000000 00000000 00000000 00000000
VR16..... 00000000 00000000 00000000 00000000        VR17..... 00000000 00000000 00000000 00000000
VR18..... 00000000 00000000 00000000 00000000        VR19..... 00000000 00000000 00000000 00000000
VR20..... 00000000 00000000 00000000 00000000        VR21..... 00000000 00000000 00000000 00000000
VR22..... 00000000 00000000 00000000 00000000        VR23..... 00000000 00000000 00000000 00000000
VR24..... 00000000 00000000 00000000 00000000        VR25..... 00000000 00000000 00000000 00000000
VR26..... 00000000 00000000 00000000 00000000        VR27..... 00000000 00000000 00000000 00000000
VR28..... 00000000 00000000 00000000 00000000        VR29..... 00000000 00000000 00000000 00000000
VR30..... 00000000 00000000 00000000 00000000        VR31..... 00000000 00000000 00000000 00000000
GPREG STORAGE:

Information for enclave COBDUMP1
Information for thread 8890000008000000
Registers on Entry to CEE3DMP:
PM....... 0000
```

Debugging COBOL programs 227
Finding COBOL information in a dump

Like the standard Language Environment dump format, dumps generated from COBOL programs contain:

- Control block information for active programs
- Storage for each active program
• Enclave-level data
• Process-level data

Control block information for active routines

The Control Blocks for Active Routines section of the dump, shown in Figure 90 on page 230, 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. For Enterprise COBOL V5.1 and later releases, the TGT is replaced by COBDSACB. The CLLE is a COBOL control block that is allocated by the COBOL runtime for each program. The CLLE is dumped for IBM service personnel use.
Storage for each active routine

The Storage for Active Routines section of the dump, shown in “Storage for active COBOL programs” on page 231, displays the following information for each COBOL program:

- Program name
- Contents of the base locators for files, WORKING-STORAGE, LINKAGE SECTION, LOCAL-STORAGE SECTION, variably-located areas, and EXTERNAL data.
- File record contents.
- WORKING-STORAGE, including the base locator for WORKING-STORAGE (BLW) and program class storage.
## Storage for active COBOL programs

### Storage for Active Routines:
**COBDUMP2**
- Contents of base locators for files are: 0-0001E038
- Contents of base locators for WORKING-STORAGE are: 0-114A411D8
- Contents of base locators for the LINKAGE SECTION are: 0-00000000 1-114A4112D

No indexes were used in this program.
No variably-located areas were used in this program.
No EXTERNAL data was used in this program.
No OBJECT instance data were used in this program.
No LOCAL-STORAGE was used in this program.
No DSA indexes were used in this program.
No FACTORY data was used in this program.
No XML data was used in this program.

### File record contents for COBDUMP2
**ESDS1D (BLF-O):** 0001E038
- Program class storage: 11480C40

**WORKING-STORAGE for COBDUMP2**
- BLW-0: 114A411D8
- same as above
- BLW-1: 114A4112D
- Program class storage: 11480C40

**LINKAGE SECTION for COBDUMP2**
- Program class storage: 0001E028

**Program class storage:** 114A41448
- Program class storage:** 114A41438

### Enclave-level data

The Enclave Control Blocks section of the dump, shown in Figure 91 on page 232, displays the following information:
- RUNCOM control block. The RUNCOM is a control block that is allocated by the COBOL runtime to anchor enclave level resources. The RUNCOM is dumped for IBM service personnel use.

**Note:** In Enterprise COBOL V5.1 and later releases, the RUNCOM control block is replaced by the COBEBD control block.

- Storage for all run units
- COBOL control blocks FCB, FIB, and GMAREA. The FCB, FIB, and GMAREA are control blocks used for COBOL file processing. These control blocks are dumped for IBM service personnel use.

---

**Figure 91: Enclave-level data for COBOL programs**

### Enclave Control Blocks:

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUNCOM</td>
<td>11480000</td>
</tr>
<tr>
<td>THDCOM</td>
<td>11490000</td>
</tr>
<tr>
<td>COBCOM</td>
<td>11500000</td>
</tr>
<tr>
<td>COBVEC</td>
<td>11510000</td>
</tr>
<tr>
<td>ITBLK</td>
<td>11520000</td>
</tr>
</tbody>
</table>

### File Control Blocks:

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCB</td>
<td>11480100</td>
</tr>
<tr>
<td>FIB</td>
<td>11490100</td>
</tr>
<tr>
<td>GMAREA</td>
<td>11500100</td>
</tr>
</tbody>
</table>

### Storage:

- **Enclave Control Blocks:**
  - RUNCOM: 11480000
  - THDCOM: 11490000
  - COBCOM: 11500000
  - COBVEC: 11510000
  - ITBLK: 11520000

- **File Control Blocks:**
  - FCB: 11480100
  - FIB: 11490100
  - GMAREA: 11500100

- **Note:** Enclave Control Blocks include COBEBD, RUNCOM, THDCOM, COBCOM, COBVEC, and ITBLK.

**Process-level data**

The Process Control Block section of the dump, shown in Figure 92 on page 233, displays COBOL process-level control blocks THDCOM, COBCOM, COBVEC, and ITBLK. For Enterprise COBOL V5.1 and later releases, the process-level control blocks are COBPCB (corresponds to THDCOM), COBCRCB (corresponds to COBCOM) and LIBVEC (corresponds to COBVEC). The control blocks are dumped for IBM service personnel use.

In a non-CICS environment, the ITBLK control block only appears when a VS COBOL II program is active. In a CICS environment, the ITBLK control block always appears.
The following examples help demonstrate techniques for debugging COBOL programs. Important areas of the dump output are highlighted. Data unnecessary to debugging has been replaced by vertical ellipses.

**Subscript range error**

Figure 93 on page 234 illustrates the error of using a subscript value outside the range of an array. This program was compiled with LIST, TEST, and SSRANGE. The SSRANGE compiler option causes the compiler to generate code that checks (during run time) for data that has been stored or referenced outside of its defined area because of incorrect indexing and subscripting. The SSRANGE option takes effect during run time. For COBOL V4R2 and prior releases, you can disable the check by specifying the CHECK(OFF) runtime option. For Enterprise COBOL V5.1 and later releases, the CHECK runtime option is ignored.

The program was run with TERMTHDACT(TRACE) to generate the traceback information shown in “Sections of Language Environment dump for COBOLX” on page 235.
To understand the traceback information and debug this program, use the following steps:

1. Locate the current error message in the Condition Information for Active Routines section of the Language Environment traceback, shown in “Sections of Language Environment dump for COBOLX” on page 235. The message is IGZ0006S The reference to table SLOT by verb number 01 on line 000011 addressed an area outside the region of the table. The message indicates that line 11 was the current COBOL statement when the error occurred. For more information about this message, see z/OS Language Environment Runtime Messages.

2. Statement 11 in the traceback section of the dump occurred in program COBOLX.

3. Find the statement on line 11 in the listing for program COBOLX, shown in Figure 94 on page 234. This statement moves the 1 value to the array SLOT (J).

4. Find the values of the local variables in the Parameters, Registers, and Variables for Active Routines section of the traceback, shown in “Sections of Language Environment dump for COBOLX” on page 235. J, which is of type PIC 9(4) with usage COMP, has a 9 value. J is the index to the array SLOT.

The array SLOT contains eight positions. When the program tries to move a value into the J or 9th element of the 8-element array named SLOT, the error of moving a value outside the area of the array occurs.
Sections of Language Environment dump for COBOLX

Figure 95 on page 236 demonstrates the error of calling a nonexistent subroutine in a COBOL program. In this example, the program COBOLX was compiled with the compiler options LIST, MAP and XREF. The TEST option was also specified with the suboptions NONE and SYM. Figure 95 on page 236 shows the program.
CBL LIST,MAP,XREF,TEST(NONE,SYM)
ID DIVISION.
PROGRAM-ID. COBOLY.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-Storage SECTION.
77  SUBNAME PIC X(8) USAGE DISPLAY VALUE 'UNKNOWN'.
PROCEDURE DIVISION.
   CALL SUBNAME.
   GOBACK.

Figure 95: COBOL example of calling a nonexistent subroutine

To understand the traceback information and debug this program, use the following steps:

1. Locate the error message for the original condition under the Condition Information for Active Routines section of the dump, shown in Figure 96 on page 237. The message is CEE3501S The module UNKNOWN was not found. For more information about this message, see z/OS Language Environment Runtime Messages.

2. Note the sequence of calls in the Traceback section of the dump. 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.

Note: If COBOLY is compiled with Enterprise COBOL V5.1 or later releases, the traceback section of Language Environment dump is shown in Figure 97 on page 238. The only difference is that IGZCFCC and IGZCLDL are combined and replaced by IGZXFCA1. (IGZXFCA1 attempts to load a non-existent routine.)
Figure 96: Sections of Language Environment dump for COBOLY
3. Use the offset of X'36E' from the COBOL listing, shown below, 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.

```cobol
DQA Entry  E Offset Statement  Load Mod  Program Unit  Service  Status
1  CEEHDSP  +0000001CA  8          COBOLY               COBOLY                                  Call
2  CEEHSGLT +000000600  8          CEEPLPKA             CEEHSGLT                       HLE7780  Exception
3  IGZXFCA1 +000009930          8          IGZALPKA             IGZXFCA1                        Call
4  COBOLY   +000001CA  B          COBOLY               COBOLY                                  Call

Figure 97: Portion of traceback of Language Environment dump for COBOLY (when compiled with Enterprise COBOL V5.1 or later)
4. Find the value of the local variables in the Parameters, Registers, and Variables for Active Routines section of the dump, shown in Figure 98 on page 239. 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.

<table>
<thead>
<tr>
<th>Parameters, Registers, and Variables for Active Routines:</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBOLY (DSA address 1147E030):</td>
</tr>
<tr>
<td>UPSTACK</td>
</tr>
<tr>
<td>DSA</td>
</tr>
<tr>
<td>Saved Registers:</td>
</tr>
<tr>
<td>GPREG0...... 1147E1B0 GPRI0...... 1147E128 GPRI2...... 0.00197FC GPRI3...... 0.00083FC</td>
</tr>
<tr>
<td>GPRI4...... 0.00082FC GPRI5...... 11.208778 GPRI6...... 0.0000000 GPRI7...... 0.0000000</td>
</tr>
<tr>
<td>GPRI8...... 0.0000000 GPRI9...... 0.0000000 GPRI10...... 0.0000000 GPRI11...... 0.0000000</td>
</tr>
<tr>
<td>GPRI12...... 0.00004EC GPRI13...... 11.47E030 GPRI14...... 0.0008760 GPRI15...... 0.0000000</td>
</tr>
<tr>
<td>GPREG</td>
</tr>
<tr>
<td>STORAGE:</td>
</tr>
<tr>
<td>Storage around GPRI0 (1147E130)</td>
</tr>
<tr>
<td>-0.020 11.47E160 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
To debug this application, use the following steps:

1. Locate the error message for the current condition in the Condition Information section of the dump, shown in “Sections of Language Environment dump for program COBOLZ1” on page 243. The message is CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).

For additional information about this message, see z/OS Language Environment Runtime Messages.

2. Note the sequence of calls in the call chain. COBOLZ1 called IGZCFCC, which is a COBOL library subroutine used for dynamic calls; IGZCFCC called COBOLZ2; COBOLZ2 then called IGZCFCC; and IGZCFCC called ASSEMZ3. The exception occurred at this point, resulting in a call to CEEHDSP, a Language Environment condition handling routine.

The call to ASSEMZ3 occurred at statement 11 of COBOLZ2. The exception occurred at offset +64 in ASSEMZ3.
3. Locate statement 11 in the COBOL listing for the COBOLZ2 program, shown in Figure 100 on page 241. This is a call to the assembler routine ASSEMZ3.

4. Check offset +64 in the listing for the assembler routine ASSEMZ3, shown in Figure 101 on page 242. 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.
5. Check local variables for COBOLZ2 in the Local Variables section of the dump shown in Figure 102 on page 242. From the dump and listings, you know that COBOLZ2 called ASSEMZ3 and passed a parameter in the variable DV-VAL. The two variables DV-VAL and D-VAL have 0 values.
6. In the COBOLZ2 subroutine, the variable D-VAL is moved to DV-VAL, the parameter passed to the assembler routine. D-VAL appears in the Linkage section of the COBOLZ2 listing, shown in Figure 103 on page 243, indicating that the value did pass from COBOLZ1 to COBOLZ2.

```cobol
000001 DIVISION.
000002 PROGRAM-ID. COBOLZ2.
000003 ENVIRONMENT DIVISION.
000004 DATA DIVISION.
000005 WORKING-STORAGE SECTION.
000006 77 D-VAL PIC 9(4) USAGE COMP.  BLW=00000+000 2C
000007 LINKAGE SECTION.
000008 77 D-VAL PIC 9(4) USAGE COMP.  BLL=00001+000 2C
000009 PROCEDURE DIVISION USING D-VAL.
000010 MOVE D-VAL TO DV-VAL.  8 6
000011 CALL "ASSEMZ3" USING DV-VAL.  EXT 6
000012 GOBACK.
/* COBOLZ2 */
```

Figure 103: Listing for COBOLZ2

7. In the Local Variables section of the dump for program COBOLZ1, shown in Figure 104 on page 243, 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.

```plaintext
Local Variables:
6 77 D-VAL            9999 COMP        00000
```

Figure 104: Variables section of Language Environment dump for COBOLZ1
Parameters, Registers, and Variables for Active Routines:

COBOLZ2 (DSA address 1147E3C0):
- UPSTACK DSA
- Saved Registers:
  - GPR0: 1147E570
  - GPR1: 1147E4C0
  - GPR2: 000197FC
  - GPR3: 000212E8
  - GPR4: 1147E4B8
  - GPR5: 000191BC
  - GPR6: 1147AFD0
  - GPR7: 00FDD100
  - GPR8: 000213A8
  - GPR9: 000211B0
  - GPR10: 0001F428
  - GPR11: 0001F54C
  - GPR12: 0001F41C
  - GPR13: 1147E3C0
  - GPR14: 8001F5C6
  - GPR15: 9140A5F8

GPREG STORAGE:
- Storage around GPR0 (1147E570)
  - -0020 1147E550
  - +0000 1147E570
  - +0020 1147E590

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

COBOLZ1 (DSA address 1147E030):
- UPSTACK DSA
- Saved Registers:
  - GPR0: 1147E1E0
  - GPR1: 1147E130
  - GPR2: 000197FC
  - GPR3: 0006A2E4
  - GPR4: 1147E128
  - GPR5: 1120B778
  - GPR6: 00000000
  - GPR7: 00000000
  - GPR8: 0000A3A8
  - GPR9: 0000A1B0
  - GPR10: 000084D8
  - GPR11: 000085F4
  - GPR12: 000084CC
  - GPR13: 1147E030
  - GPR14: 80008660
  - GPR15: 9140A5F8

GPREG STORAGE:
- Storage around GPR0 (1147E1E0)
  - -0020 1147E1C0
  - +0000 1147E1E0
  - +0020 1147E200

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

This section provides information to help you debug applications that contain one or more Fortran routines. It includes the following topics:

• Determining the source of errors in Fortran routines
• Using Fortran compiler listings
• Generating a Language Environment dump of a Fortran routine
• Finding Fortran information in a dump
• Examples of debugging Fortran routines

Determining the source of errors in Fortran routines

Most errors in Fortran routines can be identified by the information provided in Fortran runtime messages, which begin with the prefix "FOR". The Fortran compiler cannot identify all possible errors. The following list identifies several errors not detected by the compiler that could potentially result in problems:

• Failing to assign values to variables and arrays before using them in your program.
• Specifying subscript values that are not within the bounds of an array. If you assign data outside the array bounds, you can inadvertently destroy data and instructions.
• Moving data into an item that is too small for it, resulting in truncation.
• Making invalid data references to EQUIVALENCE items of differing types (for example, integer or real).
• Transferring control into the range of a DO loop from outside the range of the loop. The compiler issues a warning message for all such branches if you specify OPT(2), OPT(3), or VECTOR.
• Using arithmetic variables and constants that are too small to give the precision you need in the result. For example, to obtain more than 6 decimal digits in floating-point results, you must use double precision.
• Concatenating character strings in such a way that overlap can occur.
• Trying to access services that are not available in the operating system or hardware.
• Failing to resolve name conflicts between Fortran and C library routines using the procedures described in z/OS Language Environment Programming Guide.

Identifying runtime errors

Fortran has several features that help you find runtime errors. Fortran runtime messages are discussed in z/OS Language Environment Runtime Messages. Other debugging aids include the optional traceback map, program interruption messages, abnormal termination dumps, and operator messages.

• The optional traceback map helps you identify where errors occurred while running your application. The TERMTHDACT(TRACE) runtime option, which is set by default under Language Environment, generates a dump containing the traceback map.

You can also get a traceback map at any point in your routine by invoking the ERRTRA subroutine.

• Program interruption messages are generated whenever the program is interrupted during execution. Program interruption messages are written to the Language Environment message file.

The program interruption message indicates the exception that caused the termination; the completion code from the system indicates the specification or operation exception resulting in termination.

• Program interruptions causing an abnormal termination produce a dump, which displays the completion code and the contents of registers and system control fields.
To display the contents of main storage as well, you must request an abnormal termination (ABEND) dump by including a SYSUDUMP DD statement in the appropriate job step. The following example shows how the statement can be specified for IBM-supplied cataloged procedures:

```
//GO.SYSUDUMP DD SYSOUT=A
```

- You can request various dumps by invoking any of several dump service routines while your program runs. These dump service routines are discussed in “Generating a Language Environment dump of a Fortran routine” on page 247.

- Operator messages are displayed when your program issues a PAUSE or STOP $n$ statement. These messages help you understand how far execution has progressed before reaching the PAUSE or STOP statement.

The operator message can take the following forms:

- **$n$**
  - String of 1–5 decimal digits you specified in the PAUSE or STOP statement. For the STOP statement, this number is placed in R15.

- **'message'**
  - Character constant you specified in the PAUSE or STOP statement.

- **0**
  - Printed when a PAUSE statement containing no characters is executed (not printed for a STOP statement).

A PAUSE message causes the program to stop running pending an operator response. The format of the operator's response to the message depends on the system being used.

- Under Language Environment, error messages produced by Language Environment and Fortran are written to a common message file. Its ddname is specified in the MSGFILE runtime option. The default ddname is SYSOUT.

Fortran information directed to the message file includes:

- Error messages resulting from unhandled conditions
- Printed output from any of the dump services (SDUMP, DUMP/PDUMP, CDUMP/CPDUMP)
- Output produced by a WRITE statement with a unit identifier having the same value as the Fortran error message unit
- Output produced by a WRITE statement with * given as the unit identifier (assuming the Fortran error message unit and standard print unit are the same unit)
- Output produced by the PRINT statement (assuming the Fortran error message unit and the standard print unit are the same unit)

For more information about handling message output using the Language Environment MSGFILE runtime option, see z/OS Language Environment Programming Guide.

---

### Using Fortran compiler listings

Fortran listings provide you with:

- The date of compilation including information about the compiler
- A listing of your source program
- Diagnostic messages telling you of errors in the source program
- Informative messages telling you the status of the compilation

Table 41 on page 247 lists of the contents of the various compiler-generated listings that you might find helpful when you use information in dumps to debug Fortran programs.
Table 41: 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**

```fortran
CALL {DUMP | PDUMP} (a_1, b_1,k_1, a_2,b_2, k_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 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 \texttt{TABLE} is an array of 20 elements. The following call to the storage dump routine could be used to dump \texttt{TABLE} and \texttt{B} in hexadecimal format, and stop the program after the dump is taken.

\begin{verbatim}
CALL DUMP(TABLE(1),TABLE(20),0,B,B,0)
\end{verbatim}

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 \texttt{A} and \texttt{B} in REAL*8 format without stopping the program.

\begin{verbatim}
CALL PDUMP(A,A,6,B,B,6)
\end{verbatim}

If variables not in common are to be dumped, each variable must be listed separately in the argument list. For example, if \texttt{R}, \texttt{P}, and \texttt{Q} are defined implicitly in the program, the following statement should be used to dump the three variables in REAL*4 format.

\begin{verbatim}
CALL PDUMP(R,R,5,P,P,5,Q,Q,5)
\end{verbatim}

If the following statement is used, all main storage between \texttt{R} and \texttt{Q} is dumped, which might or might not include \texttt{P}, and could include other variables.

\begin{verbatim}
CALL PDUMP(R,Q,5)
\end{verbatim}

\textbf{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.

\textbf{Syntax}

\begin{verbatim}
CALL {CDUMP | CPDUMP} (a_1, b_1, a_2, b_2,...)
\end{verbatim}

\textit{a} and \textit{b}

Variables in the program unit. Each indicates an area of storage to be dumped. Either \textit{a} or \textit{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.

\textbf{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 \texttt{B} is a character variable and \texttt{TABLE} is a character array of 20 elements. The following call to the storage dump routine could be used to dump \texttt{TABLE} and \texttt{B} in character format, and stop the program after the dump is taken.

\begin{verbatim}
CALL CDUMP(TABLE(1), TABLE(20), B, B)
\end{verbatim}

\textbf{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 \textit{VS FORTRAN Version 2 Programming Guide for CMS and MVS}.

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

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

Syntax

CALL SDUMP [(rtn1,rtn2,...)]

rtn1,rtn2,...
Names of other program units from which data will be dumped. These names must be listed in an EXTERNAL statement.

Usage considerations for SDUMP

• To obtain symbolic dump information and location of error information, compilation must be done either with the SDUMP option or with the TEST option.
• Calling SDUMP and specifying program units that have not been entered gives unpredictable results.
• Calling SDUMP with no parameters produces the symbolic dump for the current program unit.
• An EXTERNAL statement must be used to identify the names being passed to SDUMP as external routine names.
• At higher levels of optimization (1, 2, or 3), the symbolic dump could show incorrect values for some variables because of compiler optimization techniques.
• Values for uninitialized variables are unpredictable. Arguments in uncalled subprograms or in subprograms with argument lists shorter than the maximum can cause the SDUMP subroutine to fail.
• The display of data can also be invoked automatically. If the runtime option TERMTHDACT(DUMP) is in effect and your program abends in a program unit compiled with the SDUMP option or with the TEST option, all data in that program unit is automatically dumped. All data in any program unit in the save area traceback chain compiled with the SDUMP option or with the TEST option is also dumped. Data occurring in a common block is dumped at each occurrence, because the data definition in each program unit could be different.

Examples of calling SDUMP from the main program and from a subprogram follow. Figure 105 on page 251 shows a sample program calling SDUMP and Figure 107 on page 252 shows the resulting output that is generated. In the main program, the following statement

EXTERNAL PGM1, PGM2, PGM3

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

CALL SDUMP (PGM1, PGM2, PGM3)

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

In the subprogram PGM1, the following statement makes PGM2 and PGM3 available. (PGM1 is missing because the call is in PGM1.)

EXTERNAL PGM2, PGM3

The following statements dump the variables PGM1, PGM2, and PGM3.

CALL SDUMP
CALL SDUMP (PGM2, PGM3)
Figure 105: Example program that calls SDUMP

```fortran
PROGRAM FORTMAIN
EXTERNAL PGM1, PGM2, PGM3
INTEGER*4 ANY_INT
INTEGER*4 INT_ARR(3)
CHARACTER*20 CHAR_VAR
ANY_INT = 555
INT_ARR(1) = 1111
INT_ARR(2) = 2222
INT_ARR(3) = 2222
CHAR_VAR = 'SAMPLE CONSTANT '
CALL PGM1(ANY_INT, CHAR_VAR)
CALL SDUMP(PGM1, PGM2, PGM3)
STOP
END
```

Figure 106: Example program that calls SDUMP (continued)

```fortran
SUBROUTINE PGM1(ARG1, ARG2)
EXTERNAL PGM2, PGM3
INTEGER*4 ARG1
CHARACTER*20 ARG2
ARG1 = 1
ARG2 = 'ARGUMENT'
CALL PGM2
CALL SDUMP(PGM2, PGM3)
RETURN
END
```

```fortran
SUBROUTINE PGM2
INTEGER*4 PGM2VAR
PGM2VAR = 555
CALL PGM3
RETURN
END
```

```fortran
SUBROUTINE PGM3
CHARACTER*20 PGM3VAR
PGM3VAR = 'PGM3 VAR'
RETURN
END
```

Debugging Fortran routines 251
<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
<th>Local Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM1</td>
<td>ARG2 CHARACTER*20 ARGUMENT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARG1 INTEGER*4 1</td>
<td></td>
</tr>
<tr>
<td>PGM2</td>
<td>PGM2VAR INTEGER*4 555</td>
<td></td>
</tr>
<tr>
<td>PGM3</td>
<td>PGM3VAR CHARACTER*20 PGM3 VAR</td>
<td></td>
</tr>
<tr>
<td>PGM1</td>
<td>ARG2 CHARACTER*20 ARGUMENT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ARG1 INTEGER*4 1</td>
<td></td>
</tr>
<tr>
<td>PGM2</td>
<td>PGM2VAR INTEGER*4 555</td>
<td></td>
</tr>
<tr>
<td>PGM3</td>
<td>PGM3VAR CHARACTER*20 PGM3 VAR</td>
<td></td>
</tr>
</tbody>
</table>

Figure 107: Language Environment dump generated using SDUMP

Finding Fortran information in a Language Environment dump

To locate Fortran-specific information in a Language Environment dump, you must understand how to use the traceback section and the section in the symbol table dump showing parameters and variables. Figure 108 on page 253 shows an example of a Fortran dump; Table 42 on page 253 provides additional information about each section within the dump.
Information for enclave SAMPLE

Information for thread 0000000000000000

[1]

Traceback:

<table>
<thead>
<tr>
<th>DSA Addr</th>
<th>Program Unit</th>
<th>PU Addr</th>
<th>PU Offset</th>
<th>Entry</th>
<th>E Addr</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002D018</td>
<td>CEEHDSP</td>
<td>05936760</td>
<td>+0000277C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CEEPLPKA</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>0002F018</td>
<td>AFHCSGLE</td>
<td>059DF718</td>
<td>+000001A8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AFHPRNAG</td>
<td>Exception</td>
<td></td>
</tr>
<tr>
<td>05A44060</td>
<td>AFHDPNPR</td>
<td>05A11637</td>
<td>+00001EDE</td>
<td>AFHDPNPR</td>
<td>05A11637</td>
<td>+00001EDE</td>
<td>AFHPRNAG</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05900A90</td>
<td>SAMPLE</td>
<td>059009A8</td>
<td>+0000021C</td>
<td>SAMPLE</td>
<td>059009A8</td>
<td>+0000021C</td>
<td></td>
<td>6_ISN</td>
<td>GO</td>
<td>Call</td>
</tr>
</tbody>
</table>

[2]

Condition Information for Active Routines

Condition Information for AFHCSGLE (DSA address 0002F018)

CIB Address: 0002D468

Current Condition:

FOR1916S The OPEN statement for unit 999 failed. The unit number was either less than 0 or greater than 99, the highest unit number allowed at your installation.

Location:

Program Unit: AFHCSGLE Entry: AFHCSGLE Statement: Offset: +000001A8

Storage dump near condition, beginning at location: 059DF8B0

+000000 059DF8B0  5060D198 5880C2B8 58F0801C 4110D190  05EFD502 D3019751 4770A1F0 4820D2FE  |&-

Parameters, Registers, and Variables for Active Routines:

CEEHDSP (DSA address 0002D018):

Saved Registers:

GPR0..... 000003E7  GPR1..... 0002D3B4  GPR2..... 0002DFD7  GPR3..... 0002E027
GPR4..... 0002DF94  GPR5..... 00000000  GPR6..... 00000004  GPR7..... 00000000
GPR8..... 0002E017  GPR9..... 0593875E  GPR10.... 0593775F  GPR11.... 85936760
GPR12.... 00014770  GPR13.... 0002D018  GPR14.... 800250DE  GPR15.... 85949C70

GPREG STORAGE:

Storage around GPR0 (000003E7)

-000020 00000006 00000000 0002E017 0593775E 0593775F 05936760 00014770 00000000

Figure 108: Sections of the Language Environment dump

Table 42 on page 253 describes the sections shown in the sample dump in Figure 108 on page 253.

Table 42: Understanding the Language Environment traceback table

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>The traceback section of the dump contains condition information about your routine and information about the statement number and address where the exception occurred. The traceback section helps you locate where an error occurred in your program. The information in this section begins with the most recent program unit and ends with the first program unit.</td>
</tr>
<tr>
<td>[2]</td>
<td>The condition information section contains information for the active routines. It indicates the program message, program unit name, the statement number, and the offset within the program unit where the error occurred.</td>
</tr>
</tbody>
</table>
Table 42: Understanding the Language Environment traceback table (continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3]</td>
<td>The local variable section contains information on all variables and arrays in each program unit in the save area chain, including the program that caused the dump to be invoked. The output shows variable items (one line only) and array (more than one line) items. Use the local variable section of the dump to identify the variable name, type, and value at the time the dump was called. Variable and array items can contain either character or noncharacter data, but not both.</td>
</tr>
<tr>
<td>[4]</td>
<td>The file status and attribute section of the dump displays the total number of units defined, the default units for error messages, and the default unit numbers for formatted input or formatted output.</td>
</tr>
</tbody>
</table>

Examples of debugging Fortran routines

This section contains examples of Fortran routines and instructions for using information in the Language Environment dump to debug them.

Calling a nonexistent routine

Figure 109 on page 254 illustrates an error caused by calling a nonexistent routine. The options in effect at compile time appear at the top of the listing.

```plaintext
OPTIONS IN EFFECT: LIST NOMAP NOXREF NOGOSTMT NOCODE SOURCE TERM OBJECT FIXED TRMFLG SRCFLG NODIM NORENT SDUMP(ISN) NOSXM NOVECTOR IL(DIM) NOTEST SC(*) NOEC NOICA NODIRECTIVE NOOBCS NOAAA NOPARALLEL NODYNAM NOSYM NOREORDER NOPC OPT(0) LANGLVL(77) NOFIPS FLAG(I) AUTODBL(NONE) PTRSIZE(8) LINECOUNT(60) CHARLEN(500) NAME(MAIN#)

1         PROGRAM CALLNON
2         INTEGER*4 ARRAY_END
3         CALL SUBNAM
4         STOP
5         END
```

Figure 109: Example of calling a nonexistent routine

Figure 110 on page 255 shows sections of the dump generated by a call to SDUMP.
To understand the traceback section, and debug this example routine, do the following:

1. Find the Current Condition information in the Condition Information for Active Routines section of the dump. The message is CEE3201S. The system detected an operation exception at statement 3. For more information about this message, see z/OS Language Environment Runtime Messages. This section of the dump also provides such information as the name of the active routine and the current statement number at the time of the dump.

2. Locate statement 3 in the routine shown in Figure 109 on page 254. This statement calls subroutine SUBNAM. The message CEE3201S in the Condition Information section of the dump indicates that the operation exception was generated because of an unresolved external reference.

3. Check the linkage editor output for error messages.

**Divide-by-zero error**

Figure 111 on page 256 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.
PROGRAM DIVZERO
  INTEGER*4 ANY_NUMBER
  INTEGER*4 ANY_ARRAY(3)
  PRINT *, 'EXAMPLE STARTING'
  ANY_NUMBER = 0
  DO I = 1,3
    ANY_ARRAY(I) = I
  END DO
  CALL DIVZEROSUB(ANY_NUMBER, ANY_ARRAY)
  PRINT *, 'EXAMPLE ENDING'
  STOP
END

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

Figure 111: Fortran routine with a divide-by-zero error

Figure 112 on page 257 shows the Language Environment dump for routine DIVZERO.
Information for enclave DIVZERO

Information for thread 8000000000000000

Traceback:

<table>
<thead>
<tr>
<th>DSA Addr</th>
<th>Program Unit</th>
<th>PU Addr</th>
<th>PU Offset</th>
<th>Entry</th>
<th>E Addr</th>
<th>E Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>00020018</td>
<td>CEEHDSP</td>
<td>05936760</td>
<td>+0000277C</td>
<td></td>
<td>05936760</td>
<td>+0000277C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05900640</td>
<td>DIVZSUB</td>
<td>05900658</td>
<td>+00006258</td>
<td>DIVZSUB</td>
<td>05900658</td>
<td>+00006258</td>
<td>5_ISN</td>
<td>GO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002F018</td>
<td>AFHLCLNR</td>
<td>0001B150</td>
<td>+00000000</td>
<td>AFHLCLNR</td>
<td>0001B150</td>
<td>+00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>059002E8</td>
<td>DIVZERO</td>
<td>05900200</td>
<td>+00002298</td>
<td>DIVZERO</td>
<td>05900200</td>
<td>+00002298</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Condition Information for Active Routines

Condition Information for DIVZSUB (DSA address 05900640)

CIB Address: 0002D468

Current Condition:

CEE3209S The system detected a fixed-point divide exception.

Location:

Program Unit: DIVZSUB
Entry: DIVZSUB
Statement: 5_ISN Offset: +00000258

Machine State:

ILC..... 0004    Interruption Code..... 0009
PSW..... 078D2A00 859007B4
GPR0..... 00000000  GPR1..... 0002D3B4  GPR2..... 0002DFD7  GPR3..... 0002E027
GPR4..... 0002DF94  GPR5..... 00000000  GPR6..... 00000004  GPR7..... 00000000
GPR8..... 0002E017  GPR9..... 0593875E  GPR10.... 0593775F  GPR11.... 05936760
GPR12.... 00014770  GPR13.... 0002D018  GPR14.... 800250DE  GPR15.... 85949C70
GPRREG STORAGE:

Storage around GPR0 (00000000)

| +000000 00000000 | Inaccessible storage. |
| +000020 00000020 | Inaccessible storage. |
| +000040 00000040 | Inaccessible storage. |

Storage around GPR1 (0002D3B4)

| -000020 0002D394 | 00000006 00000000 | 0002E017 0593875E 0593775F 05936760 0014770 00000000 |
| +000000 0002D3B4 | 0002DF94 | 0002E017 0593875E 0593775F 05936760 0014770 00000000 |
| +000000 0002D3B4 | 0002DF94 | 0002E017 0593875E 0593775F 05936760 0014770 00000000 |

Local Variables:

<table>
<thead>
<tr>
<th>I</th>
<th>INTEGER+4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY_ARRAY(3)</td>
<td>INTEGER+4</td>
</tr>
<tr>
<td>ANY_ARRAY(1)</td>
<td>1 2 3</td>
</tr>
<tr>
<td>ANY_NUMBER</td>
<td>INTEGER+4</td>
</tr>
</tbody>
</table>

File Status and Attributes:

The total number of units defined is 100.
The default unit for the PUNCH statement is 7.
The default unit for the Fortran error messages is 6.
The default unit for formatted sequential output is 6.
The default unit for formatted sequential input is 5.

Figure 112: Language Environment dump from divide-by-zero Fortran example

To debug this application, do the following:

1. Locate the error message, CEE3209S, for the current condition in the Condition Information section of the dump, shown in Figure 112 on page 257. The system detected a fixed-point divide exception. See z/OS Language Environment Runtime Messages for additional information about this message.

2. Note the sequence of the calls in the call chain:
   a. DIVZERO called AFHLCLNR, which is a Fortran library subroutine.
   b. AFHLCLNR called DIVZEROSUB.

   **Note:** When a program-unit name is longer than 7 characters, the name as it appears in the dump consists of the first 4 and last 3 characters concatenated together.

   c. DIVZEROSUB attempted a divide-by-zero operation at statement 5.
d. This resulted in a call to CEEHDSP, a Language Environment condition handling routine.
3. Locate statement 5 in the Fortran listing for the DIVZEROSUB subroutine in Figure 112 on page 257. This is an instruction to divide the contents of DIVIDEND(3) by DIVISOR.
4. Since DIVISOR is a parameter of subroutine DIVZEROSUB, go to the Parameters section of the dump shown in Figure 112 on page 257. The parameter DIVISOR shows a value of 0.
5. Since DIVISOR contains the value passed to DIVZEROSUB, check its value. ANY_NUMBER is the actual argument passed to DIVZEROSUB, and the dump and listing of DIVZERO indicate that ANY_NUMBER had value 0 when passed to DIVZEROSUB, leading to the divide-by-zero exception.
Chapter 7. Debugging PL/I for MVS & VM routines

This section contains information that can help you debug applications that contain one or more PL/I for MVS & VM routines. Following a discussion about potential errors in PL/I for MVS & VM routines, the first topic discusses how to use compiler-generated listings to obtain information about PL/I for MVS & VM routines, and how to use PLIDUMP to generate a Language Environment dump of a PL/I for MVS & VM routine. The last part of this section provides examples of PL/I for MVS & VM routines and explains how to debug them using information contained in the traceback information provided in the dump.

Determining the source of errors in PL/I for MVS & VM routines

Most errors in PL/I for MVS & VM routines can be identified by the information provided in PL/I runtime messages, which begin with the prefix IBM. For a list of these messages, see z/OS Language Environment Runtime Messages.

A malfunction in running a PL/I for MVS & VM routine can be caused by:

- Logic errors in the source routine
- Invalid use of PL/I for MVS & VM
- Unforeseen errors
- Invalid input data
- Compiler or runtime routine malfunction
- System malfunction
- Unidentified routine malfunction
- Overlaid storage

Logic errors in the source routine

Errors of this type are often difficult to detect because they often appear as compiler or library malfunctions. Some common errors in source routines are:

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

Invalid use of PL/I for MVS & VM

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

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

If an error is detected during run time and no ON-unit is provided in the routine to terminate the run or attempt recovery, the job terminates abnormally. However, the status of a routine at the point where the error occurred can be recorded by using an ERROR ON-unit that contains the statements. In the following example, the statement `ON ERROR SYSTEM` ensures that further errors do not result in a permanent loop.

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

Invalid input data

A routine should contain checks to ensure that any incorrect input data is detected before it can cause the routine to malfunction. Use the COPY option of the GET statement to check values obtained by stream-oriented input. The values are listed on the file named in the COPY option. If no file name is given, SYSPRINT is assumed.

Compiler or runtime routine malfunction

If you are certain that the malfunction is caused by a compiler or runtime routine error, you can either open a PMR or submit an APAR for the error. For more information about handling compiler and runtime routine malfunctions, see the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735). Meanwhile, you can try an alternative way to perform the operation that is causing the trouble. A bypass is often feasible, since the PL/I for MVS & VM language frequently provides an alternative method of performing operations.

System malfunction

System malfunctions include machine malfunctions and operating system errors. System messages identify these malfunctions and errors to the operator.

Unidentified routine malfunction

In most circumstances, an unidentified routine malfunction does not occur when using the compiler. If your routine terminates abnormally without an accompanying Language Environment runtime diagnostic message, the error causing the termination might also be inhibiting the production of a message. Check for the following:

- Your job control statements might be in error, particularly in defining data sets.
- Your routine might overwrite main storage areas containing executable instructions. This can happen if you have accidentally:
  - Assigned a value to a nonexistent array element. For example:
    ```
    DCL ARRAY(10);
    DO I = 1 TO 100;
    ARRAY(I) = VALUE;
    ```
  - Used an incorrect locator value for a locator (pointer or offset) variable. This type of error can occur if a locator value is obtained by means of record-oriented transmission. Ensure that locator values...
created in one routine, transmitted to a data set, and subsequently retrieved for use in another routine, are valid for use in the second routine.

- Attempted to free a nonbased variable. This can happen when you free a based variable after its qualifying pointer value has been changed. For example:

```pli
DCL A STATIC, B BASED (P);
ALLOCATE B;
P = ADDR (A);
FREE B;
```

- Used the SUBSTR pseudovariable to assign a string to a location beyond the end of the target string. For example:

```pli
DCL X CHAR (3);
I=3
SUBSTR (X, 2, I) = 'ABC';
```

To detect this type of error, enable the STRINGRANGE condition during compilation.

### Storage overlay problems

If you suspect an error in your PL/I for MVS & VM application is a storage overlay problem, check for the following:

1. The use of a subscript outside the declared bounds (check the SUBSCRIPTRANGE condition)
2. An attempt to assign a string to a target with an insufficient maximum length (check the STRINGSIZE condition)
3. The failure of the arguments to a SUBSTR reference to comply with the rules described for the SUBSTR built-in function (check the STRINGRANGE condition)
4. The loss of significant last high-order (left-most) binary or decimal digits during assignment to an intermediate result or variable or during an input/output operation (check the SIZE condition)
5. The reading of a variable-length file into a variable
6. The misuse of a pointer variable
7. The invocation of a Language Environment callable service with fewer arguments than are required

The first four situations are associated with the listed PL/I for MVS & VM conditions, all of which are disabled by default. If you suspect one of these problems exists in your routine, use the appropriate condition prefix on the suspected statement or on the BEGIN or PROCEDURE statement of the containing block.

The fifth situation occurs when you read a data record into a variable that is too small. This type of problem only happens with variable-length files. You can often isolate the problem by examining the data in the file information and buffer.

The sixth situation occurs when you misuse a pointer variable. This type of storage overlay is particularly difficult to isolate. There are a number of ways pointer variables can be misused:

- When a READ statement runs with the SET option, a value is placed in a pointer. If you then run a WRITE statement or another READ SET option with another pointer, you overlay your storage if you try to use the original pointer.
- When you try to use a pointer to allocate storage that has already been freed, you can also cause a storage overlay.
- When you attempt to use a pointer set with the ADDR built-in function as a base for data with different attributes, you can cause a storage overlay.

The seventh situation occurs when a Language Environment callable service is passed fewer arguments than its interface requires. The following example might cause a storage overlay because Language
Environment assumes that the fourth item in the argument list is the address of a feedback code, when in reality it could be residue data pointing anywhere in storage.

<table>
<thead>
<tr>
<th>Invalid calls</th>
<th>Valid calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL CEEDATE ENTRY OPTIONS(ASM); CALL CEEDATE(x,y,z); /* invalid */</td>
<td>DCL CEEDATE ENTRY(<em>,</em>,<em>,</em> OPTIONAL) OPTIONS(ASM); CALL CEEDATE(x,y,z,<em>); /</em> valid <em>/ CALL CEEDATE(x,y,z,fc); /</em> valid */</td>
</tr>
</tbody>
</table>

**Using PL/I for MVS & VM compiler listings**

The following sections explain how to generate listings that contain information about your routine. PL/I for MVS & VM listings show machine instructions, constants, and external or internal addresses that the linkage editor resolves. This information can help you find other information, such as variable values, in a dump of a PL/I for MVS & VM routine. The PL/I compiler listings included in the following sections are from the PL/I for MVS & VM product.

**Generating PL/I for MVS & VM listings and maps**

Table 43 on page 262 shows compiler-generated listings that you might find helpful when you use information in dumps to debug PL/I for MVS & VM routines. For more information about supported compiler options that generate listings, reference the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

**Table 43: 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>
<tr>
<td>Object code</td>
<td>Contents of the program control section in hexadecimal notation and translated into a pseudo-assembler format. To limit the size of the object code listing, specify a certain statement or range of statements to be listed; for example, LIST(20) or LIST(10,30).</td>
<td>LIST</td>
</tr>
<tr>
<td>Variable map, object code, static storage</td>
<td>Same as MAP and LIST options, plus contents of static internal and static external control sections in hexadecimal notation with comments</td>
<td>MAP and LIST</td>
</tr>
</tbody>
</table>

**Finding information in PL/I for MVS & VM listings**

Figure 113 on page 263 shows an example PL/I for MVS & VM routine that was compiled with LIST and MAP.
*PROCESS SOURCE, LIST, MAP;

SOURCE LISTING

<table>
<thead>
<tr>
<th>STMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Figure 113: PL/I for MVS & VM routine compiled with LIST and MAP

Figure 114 on page 264 shows the output generated by the LIST and MAP options for this routine, including the static storage map, variable storage map, and the object code listing. The sections following this example describe the contents of each type of listing.
Figure 114: Compiler-generated listings from example PL/I for MVS & VM routine
**Static internal storage map**

To get a complete variable storage map and static storage map, but not a complete LIST, specify a single statement for LIST to minimize the size of the listing; for example, LIST(1).

Each line of the static storage map contains the following information:

1. Six-digit hexadecimal offset.
2. Hexadecimal text, in 8-byte sections where possible.
3. Comment, indicating the type of item to which the text refers. The comment appears on the first line of the text for an item. Table 44 on page 265 lists some typical comments you might find in a static storage listing.

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

* PROCEDURE BASE

* END PROGRAM
Variable storage map
For automatic and static internal variables, the variable storage map contains the following information:

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

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

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

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

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

Every object code listing begins with the name of the external procedure. The actual entry point of the external procedure immediately follows the heading comment REAL ENTRY. The subsequent machine code is the prolog for the block, which performs block activation. The comment PROCEDURE BASE marks the end of the prolog. Following this is a translation of the first executable statement in the PL/I for MVS & VM source program. Table 45 on page 266 summarizes the comment used in the listing.
Table 45: 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>BEGIN BLOCK NUMBER n</td>
<td>Indicates the start of the begin block with number n</td>
</tr>
<tr>
<td>CALCULATION OF COMMONED EXPRESSION FOLLOWS</td>
<td>Indicates that an expression used more than once in the routine is calculated at this point</td>
</tr>
<tr>
<td>CODE MOVED FROM STATEMENT NUMBER n</td>
<td>Indicates object code moved by the optimization process to a different part of the routine and gives the number of the statement from which it originated</td>
</tr>
<tr>
<td>COMPILER GENERATED SUBROUTINE xxx</td>
<td>Indicates the start of compiler-generated subroutine xxx</td>
</tr>
<tr>
<td>CONTINUATION OF PREVIOUS REGION</td>
<td>Identifies the point at which addressing from the previous routine base recommences</td>
</tr>
<tr>
<td>END BLOCK</td>
<td>Indicates the end of a begin block</td>
</tr>
<tr>
<td>END INTERLANGUAGE PROCEDURE xxx</td>
<td>Identifies the end of an ILC procedure xxx</td>
</tr>
<tr>
<td>END OF COMMON CODE</td>
<td>Identifies the end of code used in running more than one statement</td>
</tr>
<tr>
<td>END OF COMPILER GENERATED SUBROUTINE</td>
<td>Indicates the end of the compiler-generated subroutine</td>
</tr>
<tr>
<td>END PROCEDURE</td>
<td>Identifies the end of a procedure</td>
</tr>
<tr>
<td>END PROGRAM</td>
<td>Indicates the end of the external procedure</td>
</tr>
<tr>
<td>INITIALIZATION CODE FOR xxx</td>
<td>Indicates the start of initialization code for variable xxx</td>
</tr>
<tr>
<td>INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS</td>
<td>Indicates that some of the code that follows was moved from within a loop by the optimization process</td>
</tr>
<tr>
<td>INTERLANGUAGE PROCEDURE xxx</td>
<td>Identifies the start of an implicitly generated ILC procedure xxx</td>
</tr>
<tr>
<td>METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED</td>
<td>Indicates that the order of the code following was changed to optimize the object code</td>
</tr>
<tr>
<td>ON-UNIT BLOCK NUMBER n</td>
<td>Indicates the start of an ON-unit block with number n</td>
</tr>
<tr>
<td>ON-UNIT BLOCK END</td>
<td>Indicates the end of the ON-unit block</td>
</tr>
<tr>
<td>PROCEDURE xxx</td>
<td>Identifies the start of the procedure labeled xxx</td>
</tr>
<tr>
<td>PROCEDURE BASE</td>
<td>Identifies the address loaded into the base register for the procedure</td>
</tr>
<tr>
<td>PROGRAM ADDRESSABILITY REGION BASE</td>
<td>Identifies the address where the routine base is updated if the routine size exceeds 4096 bytes and consequently cannot be addressed from one base</td>
</tr>
<tr>
<td>PROLOGUE BASE</td>
<td>Identifies the start of the prolog code common to all entry points into that procedure</td>
</tr>
<tr>
<td>REAL ENTRY</td>
<td>Precedes the actual executable entry point for a procedure</td>
</tr>
<tr>
<td>STATEMENT LABEL xxx</td>
<td>Identifies the position of source program statement label xxx</td>
</tr>
<tr>
<td>STATEMENT NUMBER n</td>
<td>Identifies the start of code generated for statement number n in the source listing</td>
</tr>
</tbody>
</table>

In certain cases, the compiler uses mnemonics (see Table 46 on page 267) to identify the type of operand in an instruction and, where applicable, follows the mnemonic by the name of a PL/I for MVS & VM variable.

Table 46: PL/I for MVS & VM mnemonics

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A..xxx</td>
<td>Address constant for xxx</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</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-</td>
</tr>
<tr>
<td></td>
<td>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 33.

**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.

```plaintext
PLIDUMP(char.-string-exp 1, char.-string-exp 2)
```

**char.-string-exp 1**
A dump options character string consisting of one or more of the following values. T, F, C, and A are the default options.

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

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

C
Continue. The routine continues after the dump.

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

F
FILE INFORMATION. A set of attributes for all open files is given. The contents of the file buffers are displayed if the B option is specified.

H
STORAGE in hexadecimal. A SNAP dump of the region is produced. A ddname of CEESNAP must be provided to direct the CEESNAP dump report.

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

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

NB
NOBLOCKS.

NF
NOFILES.

NH
NOSTORAGE.

NK
NOBLOCKS (when running under CICS).

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

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).

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

char-string-exp 2
A user-identified character string up to 80 characters long that is printed as the dump header.

PLIDUMP usage notes
If you use PLIDUMP, the following considerations apply:
- If a routine calls PLIDUMP a number of times, use a unique user-identifier for each PLIDUMP invocation. This simplifies identifying the beginning of each dump.
- In MVS or TSO, you can use ddnames of CEEDUMP, PLIDUMP, or PL1DUMP to direct dump output. If no ddname is specified, CEEDUMP is used.
- The data set defined by the PLIDUMP, PL1DUMP, or CEEDUMP DD statement should specify a logical record length (LRECL) of at least 131 to prevent dump records from wrapping.
- When you specify the H option in a call to PLIDUMP, the PL/I for MVS & VM library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of PLIDUMP results in a SNAP identifier of 0. For each successive invocation, the ID is increased by one to a maximum of 256, after which the ID is reset to 0.
- Support for SNAP dumps using PLIDUMP is provided only under MVS. SNAP dumps are not produced in a CICS environment.
  - If the SNAP does not succeed, the CEE3DMP DUMP file displays the message:

    Snap was unsuccessful

    Failure to define a CEESNAP data set is the most likely cause of an unsuccessful CEESNAP.
  - If the SNAP is successful, CEE3DMP displays the message:

    Snap was successful; snap ID = nnn

    where nnn corresponds to the SNAP identifier described above. An unsuccessful SNAP does not result in an incrementation of the identifier.
- To ensure portability across system platforms, use PLIDUMP to generate a dump of your PL/I for MVS & VM routine.

Finding PL/I for MVS & VM information in a dump
The following sections discuss PL/I-specific information located in the following sections of a Language Environment dump:
- Traceback
- Control Blocks for Active Routines
• Control Block Associated with the Thread
• File Status and Attributes

Traceback

Examine the traceback section of the dump, shown in Figure 116 on page 271, for condition information about your routine and information about the statement number and address where the exception occurred.

Figure 116: Traceback section of dump

PL/I for MVS & VM task traceback

A task traceback table is produced for multitasking programs showing the task invocation sequence (trace). For each task, the thread ID, CAA address (identified by TCA address in the dump), event variable address, task variable address, and absolute priority appear in the traceback table. An example is shown in Figure 117 on page 272.
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
Control blocks for active routines

This section shows the stack frames for all active routines, and the static storage. Use this section of the dump to identify variable values, determine the contents of parameter lists, and locate the timestamp. Figure 119 on page 273 shows this section of the dump.

<table>
<thead>
<tr>
<th>Control Blocks for Active Routines:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DSA for IBMRKDM: 20B45A88</td>
<td>Dump to identify variable values, determine the contents of parameter lists, and locate the timestamp.</td>
</tr>
<tr>
<td>DSA for ERR ON-unit: 20B45A88</td>
<td></td>
</tr>
<tr>
<td>Library Work Space: 00025670</td>
<td></td>
</tr>
<tr>
<td>Dynamic Save Area (ERR ON-unit): 20B45A88</td>
<td>Dynamic save area (ERR ON-unit): 20B45A88</td>
</tr>
<tr>
<td>DSA for IBMRERRI: 20B45000</td>
<td>DSA for IBMRERRI: 20B45000</td>
</tr>
<tr>
<td>DSA for ERR ON-unit: 20B45A88</td>
<td>Dump to identify variable values, determine the contents of parameter lists, and locate the timestamp.</td>
</tr>
<tr>
<td>Library Work Space: 00025670</td>
<td>Library Work Space: 00025670</td>
</tr>
<tr>
<td>Dynamic Save Area (ERR ON-unit): 20B45A88</td>
<td>Dynamic save area (ERR ON-unit): 20B45A88</td>
</tr>
<tr>
<td>DSA for IBMRERRI: 20B45000</td>
<td>DSA for IBMRERRI: 20B45000</td>
</tr>
</tbody>
</table>

Figure 119: Control blocks for active routines section of the dump (Part 1 of 3)
**Figure 120: Control blocks for active routines section of the dump (Part 2 of 3)**

**Figure 121: Control blocks for active routines section of the dump (Part 3 of 3)**

**Automatic variables**

To find automatic variables, use an offset from the stack frame of the block in which they are declared. This information appears in the variable storage map generated when the MAP compiler option is in effect. If you have not used the MAP option, you can determine the offset by studying the listing of compiled code instructions.

**Static variables**

If your routine is compiled with the MAP option, you can find static variables by using an offset in the variable storage map. If the MAP option is not in effect, you can determine the offset by studying the listing of compiled code.
**Based variables**

To locate based variables, use the value of the defining pointer. Find this value by using one of the methods described above to find static and automatic variables. If the pointer is itself based, you must find its defining pointer and follow the chain until you find the correct value. The following is an example of typical code for X BASED (P), with P AUTOMATIC:

```
58 60 D 0C8        L 6, P
58 E0 6 000        L 14, X
```

P is held at offset X'C8' from register 13. This address points to X.

Take care when examining a based variable to ensure that the pointers are still valid.

**Area variables**

Area variables are located using one of the methods described above, according to their storage class. The following is an example of typical code: for an area variable A declared AUTOMATIC:

```
41 60 D 0F8        LA 6, A
```

The area starts at offset X'F8' from register 13.

**Variables in areas**

To find variables in areas, locate the area and use the offset to find the variable.

**Contents of parameter lists**

To find the contents of a passed parameter list, first find the register 1 value in the save area of the calling routine's stack frame. Use this value to locate the parameter list in the dump. If R1=0, no parameters passed. For additional information about parameter lists, see the IBM Enterprise PL/I for z/OS library (www.ibm.com/support/docview.wss?uid=swg27036735).

**Timestamp**

If the TSTAMP compiler installation option is in effect, the date and time of compilation appear within the last 32 bytes of the static internal control section. The last three bytes of the first word give the offset to this information. The offset indicates the end of the timestamp. Register 3 addresses the static internal control section. If the BLOCK option is in effect, the timestamp appears in the static storage section of the dump.

**Control blocks associated with the thread**

This section of the dump, shown in Figure 122 on page 276, includes information about PL/I for MVS & VM fields of the CAA and other control block information.
Figure 122: Control blocks associated with the thread section of the dump (Part 1 of 2)

Figure 123: Control blocks associated with the thread section of the dump (Part 2 of 2)

CAA address

The address of the CAA control block appears in this section of the dump. If the BLOCK option is in effect, the complete CAA (including the PL/I for MVS & VM implementation appendage) appears separately from the body of the dump. Register 12 addresses the CAA.
File status and attribute information
This part of the dump includes the following information:
• The default and declared attributes of all open files
• Buffer contents of all file buffers
• The contents of FCBs, DCBs, DCLCBs, IOCBs, and control blocks for the process or enclave

PL/I for MVS & VM contents of the Language Environment trace table
Language Environment provides three PL/I for MVS & VM trace table entry types that contain character data:
• Trace entry 100 occurs when a task is created.
• Trace entry 101 occurs when a task that contains the tasking CALL statements is terminated.
• Trace entry 102 occurs when a task that does not contain a tasking CALL statement is terminated.

The format for trace table entries 100, 101, and 102 is shown in the following example. For more information about the Language Environment trace table format, see “Understanding the trace table entry (TTE)” on page 152.

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
The following example 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.

```plaintext
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   PAGE   2
OPTIONS(MAIN);
SOURCE
LISTING
STMT

1  EXAMPLE:  PROC
OPTIONS(MAIN):
```
The following examples show sections of the dump generated by a call to PLIDUMP.
Figure 124: Sections of the Language Environment dump (Part 1 of 2)

<table>
<thead>
<tr>
<th>Information for thread 0000000000000000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASID</strong>: 003E  <strong>Job ID</strong>: JOB21950  <strong>Job name</strong>: LEADSMP1  <strong>Step name</strong>: GO  <strong>UserID</strong>: HEALY</td>
</tr>
<tr>
<td><strong>CEEEV010</strong>: CEEEV010  <strong>Offset</strong>: +0000013A  <strong>IBMREV</strong>: IBMREV10  <strong>CEEEV010</strong>: Call  <strong>DSA Entry</strong>: E  <strong>Offset</strong>: +0000013A  <strong>Statement</strong>: CEEEV010  <strong>Load Mod</strong>: EXPL  <strong>Program Unit</strong>: CEEEV010  <strong>Service</strong>: Call</td>
</tr>
</tbody>
</table>

**PLIDUMP was called from statement number 6 at offset +000000D6 from ERR ON-unit with entry address 20900C58**

**CEE3845I CEEDUMP Processing started.**

Information for thread 0000000000000000

**PLIDUMP was called from error On-unit**

**CEE3DMP V1 R9.0: Plidump called from error On-unit**

**02/27/07 11:45:20 AM**

**Page: 1**

---

Figure 125: Sections of the Language Environment dump (Part 2 of 2)

<table>
<thead>
<tr>
<th>Information for thread 088900000000900000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASID</strong>: 003E  <strong>Job ID</strong>: JOB21950  <strong>Job name</strong>: LEADSMP1  <strong>Step name</strong>: GO  <strong>UserID</strong>: HEALY</td>
</tr>
<tr>
<td><strong>CEEEV010</strong>: CEEEV010  <strong>Offset</strong>: +0000013A  <strong>IBMREV</strong>: IBMREV10  <strong>CEEEV010</strong>: Call  <strong>DSA Entry</strong>: E  <strong>Offset</strong>: +0000013A  <strong>Statement</strong>: CEEEV010  <strong>Load Mod</strong>: EXPL  <strong>Program Unit</strong>: CEEEV010  <strong>Service</strong>: Call</td>
</tr>
</tbody>
</table>

**PLIDUMP was called from statement number 6 at offset +000000D6 from ERR ON-unit with entry address 20900C58**

**Information for enclave EXAMPLE**

**Information for thread 088900000000900000**

---

Debugging PL/I for MVS & VM routines 279
Traceback:
DSA Addr E Addr Pu Addr Pu Offset Comp Date Compile Attributes

DSA for CEEDSP: 00B42A8

00890000...00000000...00000000...00000000...00000000  |...........D.................j..|

Control Blocks for Active Routines:

Condition Information for IBMRERRI (DSA address 20B42500)

IBM0281S A prior condition was promoted to the ERROR condition.

IBM0421S  ON CODE=520  The SUBSCRIPTRANGE condition was raised.

To debug this routine, use the following steps:
1. In the dump, PLIDUMP was called by the ERROR ON-unit in statement 6. The traceback information in the dump shows that the exception occurred following statement 11.

2. Locate the Original Condition message in the Condition Information for Active Routines section of the dump. The message is IBM0421S ONCODE=520 The SUBSCRIPTRANGE condition was raised. This message indicates that the exception occurred when an array element value exceeded the subscript range value (in this case, 10). For more information about this message, see z/OS Language Environment Runtime Messages.

3. Locate statement 9 in the routine in “#unique_273/unique_273_Connect_42_pliex1” on page 277. The instruction is Array_End = 20. This statement assigns a 20 value to the variable Array_End.

4. Statement 10 begins the DO-loop instruction Do I = 1 to Array_End. Since the previous instruction (statement 9) specified that Array_End = 20, the loop in statement 10 should run until I reaches a 20 value.

   The instruction in statement 2, however, declared a 10 value for the array range. Therefore, when the I value reached 11, the SUBSCRIPTRANGE condition was raised.

The following steps provide another method for finding the value that raised the SUBSCRIPTRANGE condition.

1. Locate the offset of variable I in the variable storage map in “#unique_273/unique_273_Connect_42_pliex1” on page 277. 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 124 on page 279.

   The block located at this offset contains the value that exceeded the array range, X'B' or 11.

### Calling a nonexistent subroutine

Figure 126 on page 281 demonstrates the error of calling a nonexistent subroutine. This routine was compiled with the LIST, MAP, and GOSTMT compiler options. It was run with the TERMTHDACT(DUMP) runtime option to generate a traceback.

```
688-235 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 NOOCTIONS;
688-235 IBM PL/I for MVS & VM EXAMPLE1: PROC OPTIONS(MAIN);
SOURCE LISTING
STMT
1 EXAMPLE1: PROC OPTIONS(MAIN);
2 DCL Prog01 entry external;
3 On error
4 Begin;
5 Call plidump('TBNFS','Plidump called from error On-unit');
6 End;
7 Call prog01; /* Call external program PROG01 */
8 End Example1;
```

**Figure 126: Example of calling a nonexistent subroutine**

Figure 127 on page 282 shows the traceback and condition information from the dump.
Figure 127: Sections of the Language Environment dump (Part 1 of 2)
To understand the traceback and debug this example routine, use the following steps:

1. Find the Current Condition message in the Condition Information for Active Routines section of the dump. The message is CEE3201S. The system detected an Operation exception. For more information about this message, see z/OS Language Environment Runtime Messages.

   This section of the dump also provides such information as the name of the active routine and the current statement number at the time of the dump. The Location section indicates that the exception occurred at offset X’-20900D2C’ within entry EXAMPLE1 and that there might have been a bad branch from offset X’+000000C0’ statement 7 within entry EXAMPLE1.

2. Locate statement 7 in the routine (Figure 126 on page 281). 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 129 on page 284 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.
OPTIONS SPECIFIED

*PROCESS  GOSTMT LIST S STG TEST MAP NOOPTIONS;

SAMPLE: PROC  OPTIONS(MAIN) ;

SOURCE LISTING

STMT
1  SAMPLE: PROC  OPTIONS(MAIN) ;
2       On error
3           begin;
4           On error system; /* prevent nested error conditions */
5           Call PLIDUMP('TBC','PLIDUMP called from error ON-unit');
6           Put Data; /* Display variables */
7       End;
8       DECLARE
9           A_number    Fixed Bin(31),
10           My_Name     Char(13),
11           An_Array(3) Fixed Bin(31) init(1,3,5);
12       Put skip list('Sample Starting');
13       A_number = 0;
14       My_Name = 'Tery Gillaspy';
15       Call Sub1(a_number, my_name, an_array);
16    SUB1:  PROC(divisor, name1, Array1);
17           Declare
18               Divisor   Fixed Bin(31),
19               Name1     Char(13),
20               Array1(3) Fixed Bin(31);
21           Put skip list('Sub1 Starting');
22           Array1(1) = Array1(2) / Divisor;
23           Put skip list('Sub1 Ending');
24    End SUB1;
25      Put skip list('Sample Ending');
26  End;

STORAGE REQUIREMENTS

<table>
<thead>
<tr>
<th>BLOCK, SECTION OR STATEMENT</th>
<th>TYPE</th>
<th>LENGTH   (HEX)</th>
<th>DSA SIZE (HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*SAMPLE1</td>
<td>PROGRAM CSECT</td>
<td>1060</td>
<td>424</td>
</tr>
<tr>
<td>*SAMPLE2</td>
<td>STATIC CSECT</td>
<td>860</td>
<td>35C</td>
</tr>
<tr>
<td>SAMPLE</td>
<td>PROCEDURE BLOCK</td>
<td>428</td>
<td>1AC</td>
</tr>
<tr>
<td>BLOCK 2</td>
<td>STMT 2</td>
<td>298</td>
<td>12A</td>
</tr>
<tr>
<td>SUB1</td>
<td>PROCEDURE BLOCK</td>
<td>332</td>
<td>14C</td>
</tr>
</tbody>
</table>

Figure 129: PL/I for MVS & VM routine with a divide-by-zero error

Since variables are not normally displayed in a PLIDUMP dump, this routine included a PUT DATA statement, which generated a listing of arguments and variables used in the routine. Figure 130 on page 284 shows this output.

Figure 130: Variables from routine SAMPLE

The routine in Figure 129 on page 284 was compiled with the LIST compiler option, which generated the object code listing shown in Figure 131 on page 284.

Figure 131: Object code listing from example PL/I for MVS & VM routine

Figure 132 on page 285 shows the Language Environment dump for routine SAMPLE.
Figure 132: Language Environment dump from example PL/I for MVS & VM routine (Part 1 of 3)

Figure 133: Language Environment dump from example PL/I for MVS & VM routine (Part 2 of 3)
To understand the dump information and debug this routine, use the following steps:

1. Notice the title of the dump: PLIDUMP called from error ON-unit. This was the title specified when PLIDUMP was invoked, and it indicates that the ERROR condition was raised and PLIDUMP was called from within the ERROR ON-unit.

2. Locate the messages in the Condition Information section of the dump. There are two messages. The current condition message indicates that a prior condition was promoted to the ERROR condition. The promotion of a condition occurs when the original condition is left unhandled (no PL/I for MVS & VM ON-units are assigned to gain control). The original condition message is CEE3209S. The system detected a Fixed Point divide exception. The original condition usually indicates the actual problem. For more information about this message, see z/OS Language Environment Runtime Messages.

3. In the traceback section, note the sequence of calls in the call chain. SAMPLE called SUB1 at statement 11, and SUB1 raised an exception at statement 15, PU offset X'3CE'.

4. Find the statement in the listing for SUB1 that raised the ZERODIVIDE condition. If SUB1 was compiled with GOSTMT and SOURCE, find statement 15 in the source listing. Since the object listing was generated in this example, you can also locate the actual assembler instruction causing the exception at offset X'3CE' in the object listing for this routine, shown in Figure 131 on page 284. 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'20942400' and the value of the first parameter is X'00000000'. Thus, the exception occurred when SAMPLE passed a 0 value used as a divisor in subroutine SUB1.
Chapter 8. Debugging Enterprise PL/I routines

This topic contains information that can help you debug applications that contain one or more Enterprise PL/I routines. Following a discussion about potential errors in Enterprise PL/I routines, the first part of this information discusses how to use compiler-generated listings to obtain information about Enterprise PL/I routines, and how to use PLIDUMP to generate a Language Environment dump of an Enterprise PL/I routine. The last part of the chapter provides examples of Enterprise PL/I routines and explains how to debug them using information contained in the traceback information provided in the dump.

Determining the source of errors in Enterprise PL/I routines

Most errors in Enterprise PL/I routines can be identified by the information provided in Enterprise PL/I runtime messages, which begin with the prefix IBM. For a list of these messages, see z/OS Language Environment Runtime Messages.

A malfunction in running an Enterprise PL/I routine can be caused by:

• Logic errors in the source routine
• Invalid use of Enterprise PL/I
• Unforeseen errors
• Invalid input data
• Compiler or runtime routine malfunction
• System malfunction
• Unidentified routine malfunction
• Overlaid storage

Logic errors in the source routine

Errors of this type are often difficult to detect because they often appear as compiler or library malfunctions. Some common errors in source routines are:

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

Invalid use of Enterprise PL/I

A misunderstanding of the language or a failure to provide the correct environment for using Enterprise PL/I can result in an apparent malfunction of an Enterprise PL/I routine. Any of the following, for example, might cause a malfunction:

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

If an error is detected during run time and no ON-unit is provided in the routine to terminate the run or attempt recovery, the job terminates abnormally. However, the status of a routine at the point where the error occurred can be recorded by using an ERROR ON-unit that contains the following statements. ON ERROR SYSTEM ensures that further errors do not result in a permanent loop.

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

Invalid input data

A routine should contain checks to ensure that any incorrect input data is detected before it can cause the routine to malfunction. Use the COPY option of the GET statement to check values obtained by stream-oriented input. The values are listed on the file named in the COPY option. If no file name is given, SYSPRINT is assumed.

Compiler or runtime routine malfunction

If you are certain that the malfunction is caused by a compiler or runtime routine error, you can either open a PMR or submit an APAR for the error. Meanwhile, you can try an alternative way to perform the operation that is causing the trouble. A bypass is often feasible, since the Enterprise PL/I language frequently provides an alternative method of performing operations.

System malfunction

System malfunctions include machine malfunctions and operating system errors. System messages identify these malfunctions and errors to the operator.

Unidentified routine malfunction

In most circumstances, an unidentified routine malfunction does not occur when using the compiler. If your routine terminates abnormally without an accompanying Language Environment runtime diagnostic message, the error causing the termination might also be inhibiting the production of a message. Check for the following:

• Your job control statements might be in error, particularly in defining data sets.

• Your routine might overwrite main storage areas containing executable instructions. This can happen if you have accidentally:
  - Assigned a value to a nonexistent array element. For example:
    ```pli
    DCL ARRAY(10);
    ::
    DO I = 1 TO 100;
    ARRAY(I) = VALUE;
    ```
  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:

1. The use of a subscript outside the declared bounds (check the SUBSCRIPTRANGE condition)
2. An attempt to assign a string to a target with an insufficient maximum length (check the STRINGSIZE
condition)
3. The failure of the arguments to a SUBSTR reference to comply with the rules described for the SUBSTR
built-in function (check the STRINGRANGE condition)
4. The loss of significant last high-order (left-most) binary or decimal digits during assignment to an
intermediate result or variable or during an input/output operation (check the SIZE condition)
5. The reading of a variable-length file into a variable
6. The misuse of a pointer variable
7. The invocation of a Language Environment callable service with fewer arguments than are required

The first four situations are associated with the listed Enterprise PL/I conditions, all of which are disabled
by default. If you suspect one of these problems exists in your routine, use the appropriate condition
prefix on the suspected statement or on the BEGIN or PROCEDURE statement of the containing block.

The fifth situation occurs when you read a data record into a variable that is too small. This type of
problem only happens with variable-length files. You can often isolate the problem by examining the data
in the file information and buffer.

The sixth situation occurs when you misuse a pointer variable. This type of storage overlay is particularly
difficult to isolate. There are a number of ways pointer variables can be misused:

- When a READ statement runs with the SET option, a value is placed in a pointer. If you then run a WRITE
statement or another READ SET option with another pointer, you overlay your storage if you try to use
the original pointer.
- When you try to use a pointer to allocate storage that has already been freed, you can also cause a
storage overlay.
- When you attempt to use a pointer set with the ADDR built-in function as a base for data with different
attributes, you can cause a storage overlay.

The seventh situation occurs when a Language Environment callable service is passed fewer arguments
than its interface requires. The following example might cause a storage overlay because Language
Environment assumes that the fourth item in the argument list is the address of a feedback code, when in
reality it could be residue data pointing anywhere in storage.
Using Enterprise PL/I compiler listings

The following sections explain how to generate listings that contain information about your routine. Enterprise PL/I listings show machine instructions, constants, and external or internal addresses that the linkage editor resolves. This information can help you find other information, such as variable values, in a dump of an Enterprise PL/I routine.

Note: Enterprise PL/I shares a common compiler back-end with C/C++. The Enterprise PL/I assembler listing will, consequently, have a similar form to those from the XL C/C++ compiler.

The compiler listings included below are from the Enterprise PL/I product.

Generating Enterprise PL/I listings and maps

Table 47 on page 290 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>
<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 135 on page 291 shows the first two pages of an example Enterprise PL/I routine that was compiled with the LIST, MAP and SOURCE options.
Figure 135: Enterprise PL/I routine compiled with LIST, MAP, and SOURCE

Figure 136 on page 292 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 136: Compiler-generated listings from example Enterprise PL/I routine (Part 1 of 4)
Figure 137: Compiler-generated listings from example Enterprise PL/I routine (Part 2 of 4)
*** General purpose registers used: 1111111110001111
*** Floating point registers used: 1111111100000000
*** Size of register spill area: 512(max) 0(used)
*** Size of dynamic storage: 200
*** Size of executable code: 350
*** CSECT Offset: 72 : 0x48

0001BC 0000 0000

Constant Area

000000 0004CE7 E3D9 |..EXTR | 5655-H31 IBM(R) Enterprise PL/I for z/OS
OFFSET OBJECT CODE LINE# FILE# PSEUDO ASSEMBLY LISTING

PPA1: Entry Point Constants

000000 1CCEA166 =>F'4B33037B2' Flags
000004 0000001CB =A(PPA2-EXAMPLE)
000008 00000000 =>F'0' No PPA3
00000C 00000000 =>F'0' No EPD
000010 FF000000 =>F'2097152' Register save mask
000014 00000000 =>F'0' Member flags
000018 00000000 =>AL(144) Flags
000019 00000000 =AL3(0) Callee's DSA use/8
00001C 00000000 =>F'0' State variable location
000020 FFE00000 =>F'-2097152' Register save mask
000024 00000000 =>F'0' Member flags
000028 00000000 =>F'0' State variable location
000030 00000000 =>F'0' State variable location
000034 00000000 =>F'0' State variable location
000038 00000000 =>F'0' State variable location

PPA1 End

PPA2: Compile Unit Block

000000 0B00 3203 =>F'184562179' Flags
000004 FFFF FDF0 =A(CEESTART-PPA2)
000008 0000 0000 =>F'0' No PPA4
00000C FFFF FDF0 =A(TIMESTAMP-PPA2)
000010 0000 0000 =>F'0' No primary
000014 0200 0000 =>F'33554432' Flags

PPA2 End

5655-H31 IBM(R) Enterprise PL/I for z/OS

EXTERNAL SYMBOL DICTIONARY

NAME TYPE ID ADDR LENGTH NAME TYPE ID ADDR LENGTH
EXAMPLE1 SD 1 000000 000228 EXAMPLE2 SD 2 000000 00005C
@EXAMPLE SD 3 000000 000004 B SD 4 000000 000010
EXAMPLE LD 0 000048 000001 CEESG011 ER 5 000000
IBMQFRG ER 6 000000 IBMQOFSB ER 7 000000
IBMQEFSH ER 8 000000 CEESTART ER 9 000000
CEEMAIN SD 10 000000 00000C IBMPINPL ER 11 000000

5655-H31 IBM(R) Enterprise PL/I for z/OS

EXTERNAL SYMBOL CROSS REFERENCE

ORIGINAL NAME EXTERNAL SYMBOL NAME
EXAMPLE1 EXAMPLE1
EXAMPLE2 EXAMPLE2
EXAMPLE EXAMPLE
B B
EXAMPLE EXAMPLE
CEESG011 CEESG011
IBMQFRG IBMQFRG
IBMQOFSB IBMQOFSB
IBMQEFSH IBMQEFSH
CEESTART CEESTART
CEEMAIN CEEMAIN
IBMPINPL IBMPINPL

Figure 138: Compiler-generated listings from example Enterprise PL/I routine (Part 3 of 4)
Pseudo assembly listing

The pseudo assembly listing consists of the machine instructions and a translation of these instructions into a form that resembles assembler code. This listing always starts with a small section of non-executable data that records the date and time when the object was produced as well as the version of the compiler used to produce the object. This section ends with a service string which in the listing is followed by the build date for the compiler back-end that generated this part of the listing (and this date may be different from the build date for the compiler front-end that generated the first pages of the listing).

The majority of the pseudo assembly listing consists of the object code arranged in columns that specify for each instructions:

- Its offset.
- the instruction in object code format.
- Its associated line number.
- Its associated file number if non-zero (for example, if from an include file).
- the instruction in mnemonic format.

External symbol dictionary

The external symbol dictionary lists all the external symbols generated for this compilation. For each symbol, it also lists its linkage type and size (in hex).
External symbol cross reference

The external symbol dictionary cross reference shows for each external symbol the name that will be visible externally to the linker.

Storage offset listing

Each line of the storage offset listing contains the following information for each user variable:

- Its name.
- the number of the block in which it was declared.
- the number of the file in which it was declared.
- the number of the line in which it was declared.
- Its class (automatic, static, etc).
- Its location (as appropriate for its class).
- Its byte length in decimal.

This list is sorted by block number and then by name within each block.

Static map

Each line of the static storage map contains the following information for each internal static variable:

- Its hexadecimal offset.
- Its byte length in hex.
- Its name.

This list is sorted by the offset of the variables in static. This list of variables may also include compiler-generated variables.

Automatic map

Each line of the automatic storage map contains the following information, grouped by named block, for each automatic variable in that block:

- Its hexadecimal offset.
- Its byte length in hex.
- Its name.

These lists are sorted by the offset of the variables in automatic for each block. These lists of variables may also include compiler-generated variables.

Generating a Language Environment dump of an Enterprise PL/I routine

To generate a dump of an Enterprise PL/I routine, you can call either the Language Environment callable service CEE3DMP or PLIDUMP. For information about calling CEE3DMP, see “Generating a Language Environment dump with CEE3DMP” on page 33.

PLIDUMP syntax and options

PLIDUMP calls intermediate Enterprise PL/I library routines, which convert most PLIDUMP options to CEE3DMP options. The following list contains PLIDUMP options and the corresponding CEE3DMP option, if applicable. Some PLIDUMP options do not have corresponding CEE3DMP options, but continue to function as Enterprise PL/I default options. The list following the syntax diagram provides a description of those options.

PLIDUMP conforms to National Language Support standards. PLIDUMP can supply information across multiple Language Environment enclaves. If an application running in one enclave fetches a main
procedure (an action that creates another enclave), PLIDUMP contains information about both procedures. The syntax and options for PLIDUMP are shown below.

**Syntax**

PLIDUMP (char.-string-exp 1, char.-string-exp 2)

**char.-string-exp 1**

A dump options character string consisting of one or more of the following values. T, F, C, and A are the default options.

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

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

- **C**
  - Continue. The routine continues after the dump.

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

- **F**
  - FILE INFORMATION. A set of attributes for all open files is given. The contents of the file buffers are displayed if the B option is specified.

- **H**
  - STORAGE in hexadecimal. A SNAP dump of the region is produced. A ddname of CEESNAP must be provided to direct the CEESNAP dump report.

- **K**
  - BLOCKS (when running under CICS). The Transaction Work Area is included.
  
  **Note:** This option is not supported under Enterprise PL/I.

- **NB**
  - NOBLOCKS.

- **NF**
  - NOFILES.

- **NH**
  - NOSTORAGE.

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

- **NT**
  - NOTRACEBACK.

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

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

char.-string-exp 2
A user-identified character string up to 80 characters long that is printed as the dump header.

PLIDUMP usage notes

If you use PLIDUMP, the following considerations apply:

- If a routine calls PLIDUMP a number of times, use a unique user-identifier for each PLIDUMP invocation. This simplifies identifying the beginning of each dump.
- In MVS or TSO, you can use ddnames of CEEDUMP, PLIDUMP, or PL1DUMP to direct dump output. If no ddname is specified, CEEDUMP is used.
- The data set defined by the PLIDUMP, PL1DUMP, or CEEDUMP DD statement should specify a logical record length (LRECL) of at least 131 to prevent dump records from wrapping.
- When you specify the H option in a call to PLIDUMP, the Enterprise PL/I library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of PLIDUMP results in a SNAP identifier of 0. For each successive invocation, the ID is increased by one to a maximum of 256, after which the ID is reset to 0.
- Support for SNAP dumps using PLIDUMP is provided only under MVS. SNAP dumps are not produced in a CICS environment.
  - If the SNAP does not succeed, the CEE3DMP DUMP file displays the message:

    Snap was unsuccessful

Failure to define a CEESNAP data set is the most likely cause of an unsuccessful CEESNAP.

  - If the SNAP is successful, CEE3DMP displays the message, where nnn corresponds to the SNAP identifier described above. An unsuccessful SNAP does not result in an incrementation of the identifier.

    Snap was successful; snapshot ID = nnn

- To ensure portability across system platforms, use PLIDUMP to generate a dump of your Enterprise PL/I routine.

Finding Enterprise PL/I information in a dump

The following sections discuss Enterprise PL/I-specific information located in the following sections of a Language Environment dump:

- Traceback
- Control Blocks for Active Routines
- Control Block Associated with the Thread
- File Status and Attributes
**Traceback**

Examine the traceback section of the dump, shown in Figure 140 on page 299, for condition information about your routine and information about the statement number and address where the exception occurred.

---

### Condition information

If the dump was called from an ON-unit, the type of ON-unit is identified in the traceback as part of the entry information. For ON-units, the values of any relevant condition built-in functions (for example, ONCHAR and ONSOURCE for conversion errors) appear. In cases where the cause of entry into the ON-unit is not stated, usually when the ERROR ON-unit is called, the cause of entry appears in the condition information.

### Statement number and address where error occurred

This information, which is the point at which the condition that caused entry to the ON-unit occurred, can be found in the traceback section of the dump. If the condition occurs in compiled code, and you compiled your routine with either GOSTMT or GONUMBER, the statement numbers appear in the dump. To identify the assembler instruction that caused the error, use the traceback information in the dump to find the program unit (PU) offset of the statement number in which the error occurred. Then find that offset and the corresponding instruction in the object code listing.

### Control blocks for active routines

This section shows the stack frames for all active routines, and the static storage. Use this section of the dump to identify variable values, determine the contents of parameter lists, and locate the timestamp. Figure 141 on page 300 shows this section of the dump.
Control Blocks for Active Routines:

**Automatic variables**

To find automatic variables, use an offset from the stack frame of the block in which they are declared. This information appears in the variable storage map generated when the MAP compiler option is in effect. If you have not used the MAP option, you can determine the offset by studying the listing of compiled code instructions.

**Static variables**

If your routine is compiled with the MAP option, you can find static variables by using an offset in the variable storage map. If the MAP option is not in effect, you can determine the offset by studying the listing of compiled code.

**Based variables**

To locate based variables, use the value of the defining pointer. Find this value by using one of the methods described above to find static and automatic variables. If the pointer is itself based, you must find its defining pointer and follow the chain until you find the correct value.

The following is an example of typical code for X BASED (P), with P AUTOMATIC. P is held at offset X'C8' from register 13. This address points to X.

```
58 60 D 0CS
L 6, P
58 E0 6 000
L 14, X
```

Take care when examining a based variable to ensure that the pointers are still valid.

**Area variables**

Area variables are located using one of the methods described above, according to their storage class.
The following is an example of typical code: for an area variable A declared AUTOMATIC. The area starts at offset 'X'F8' from register 13.

41 60 D 0F8  LA 6, A

Variables in areas
To find variables in areas, locate the area and use the offset to find the variable.

Contents of parameter lists
To find the contents of a passed parameter list, first find the register 1 value in the save area of the calling routine's stack frame. Use this value to locate the parameter list in the dump. If R1=0, no parameters passed.

Control blocks associated with the thread
This section of the dump, shown in Figure 142 on page 301, includes information about Enterprise PL/I fields of the CAA and other control block information.
CAA address

The address of the CAA control block appears in this section of the dump. If the BLOCK option is in effect, the complete CAA (including the Enterprise PL/I implementation appendage) appears separately from the body of the dump. Register 12 addresses the CAA.

File status and attribute information

This part of the dump includes the following information:

- The default and declared attributes of all open files
- Buffer contents of all file buffers
- The contents of FCBs, DCBs, DCLCBs, IOCBs, and control blocks for the process or enclave

Enterprise PL/I contents of the Language Environment trace table

Language Environment provides three Enterprise PL/I trace table entry types that contain character data:

- Trace entry 100 occurs when a task is created.
- Trace entry 101 occurs when a task that contains the tasking CALL statements is terminated.
- Trace entry 102 occurs when a task that does not contain a tasking CALL statement is terminated.

The format for trace table entries 100, 101, and 102 follows. For more information about the Language Environment trace table format, see “Understanding the trace table entry (TTE)” on page 152.

---(100) NameOfCallingTask NameOfCalledTask OffsetOfCallStmt UserAggPtr CalledTaskPtr TaskVarPtr EventVarPtr PriorityPtr CallingR2-R5 CallingR12-R14
---(101) NameOfReturnTask ReturnerR2-R5 ReturnerR12-R14
---(102) NameOfReturnTask

Figure 143: Control blocks associated with the thread section of the dump (Enterprise PL/I) (Part 2 of 2)
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 144 on page 303 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.

```pli
Options Specified
Install:
Command:
Line.File Process Statements
1.0 *PROCESS GONUMBER LIST 5 STG TEST MAP;
Install:
5655-H31 IBM(R) Enterprise PL/I for z/OS EXAMPLE: PROC OPTIONS(M 2007.01.31 15:59:36 Page 2
Compiler Source
Line.File
2.0 EXAMPLE: PROC OPTIONS(MAIN);
3.0
4.0 DCL Array(10) Fixed bin(31);
5.0 DCL (I,Array_End) Fixed bin(31);
6.0 ON error
7.0 Begin;
8.0 On error system;
9.0 Call plidump('tbnfs','Plidump called from error On-unit');
10.0 End;
11.0
12.0 (subrg): /* Enable subscript range condition */
13.0 Labl1: Begin;
14.0 Array_End = 20;
15.0 DO I = 1 to Array_End; /* Loop to initialize array */
16.0 Array(I) = 2; /* Set array elements to 2 */
17.0 End;
18.0 End Labl1;
19.0 EXAMPLE: PROC OPTIONS(MAIN);
Block Name List
Number Name
1 EXAMPLE
2 _ON_Begin_7_Blk_2
3 _Begin_12_Blk_3
5655-H31 IBM(R) Enterprise PL/I for z/OS EXAMPLE: PROC OPTIONS(M 2007.01.31 15:59:36 Page 3
OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G
Timestamp and Version Information
000000  F2F0  F0F7                                       =C'2007'           Compiled Year
000004  F0F1  F3F1                                       =C'0131'           Compiled Date MMDD
000008  F1F5  F5F9  F3F6                                 =C'155936'         Compiled Time HHMMSS
00000E  F0F3  F0F6  F0F0                                 =C'030600'         Compiler Version
000014  0036  ****                                       Service String     20070122
Timestamp and Version End
OFFSET OBJECT CODE LINE# FILE# PSEUDO ASSEMBLY LISTING
5655-H31 IBM(R) Enterprise PL/I for z/OS EXAMPLE: PROC OPTIONS(M 2007.01.31 15:59:36 Page 17
IDENTIFIER DEFINITION ATTRIBUTES ------<FILE NO=>--<FILE LINE-->--
ARRAY 1-0:4 Class = automatic, Location = 192 : 0xC0(x13), Length = 40
ARRAY_END 1-0:5 Class = automatic, Location = 256 : 0x8C(x13), Length = 4
I 1-0:5 Class = automatic, Location = 232 : 0x88(x13), Length = 4

Figure 144: Example of moving a value outside an array range (Enterprise PL/I)

Figure 145 on page 304 shows sections of the dump generated by a call to PLIDUMP.
CEEDUMP V1 R9.8: Plidump called from error On-unit 01/31/07 3:59:39 PM  Page: 1
ASID: 010E  Job ID: J0009410  Job name: LEDGSMP1  Step name: GO  UserID: BARBARA
CEES3ASI CEEDUMP Processing started.
PLIDUMP was called from statement number 9 at offset +000000D2 from _ON_Begin_7_Blk_2 with entry address 11200240
Information for enclave EXAMPLE
Information for thread 8000000000000000

Traceback:
<table>
<thead>
<tr>
<th>DSA</th>
<th>Entry</th>
<th>E  Offset</th>
<th>Statement</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IBMPDUMP</td>
<td>+000002AE</td>
<td></td>
<td>IBMPEV11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>_ON_Begin_7_Blk_2</td>
<td>+000000D2</td>
<td>9</td>
<td>EXAMPLE</td>
<td>_Begin_12_Blk_3</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IBMPEONR</td>
<td>+000002A2</td>
<td></td>
<td>IBMPEV11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IBMPEBOP</td>
<td>+000004DC</td>
<td></td>
<td>IBMPEV11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CEEEV011</td>
<td>+00000132</td>
<td></td>
<td>IBMPEV11</td>
<td>CEEEV011</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>CEEHDSP</td>
<td>+000017D0</td>
<td></td>
<td>CEEPLPKA</td>
<td>CEEHDSP</td>
<td>D1908</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IBMBERRI</td>
<td>+000000AA</td>
<td></td>
<td>IBMPEV11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Condition Information for Active Routines
Condition Information for (DSA address 11A3A6B8)
CIB Address: 11A3B178
Current Condition:
IBM0281S A prior condition was promoted to the ERROR condition.
Original Condition:
IBM0421S  ONCODE=520  The SUBSCRIPTRANGE condition was raised.
Location:
Program Unit:  Entry: IBMBERRI Statement:  Offset: +000000AA
Storage dump near condition, beginning at location: 114AAAB2

Figure 145: Sections of the Language Environment dump (Part 1 of 2)

Figure 146 on page 305 shows more sections of the dump generated by a call to PLIDUMP.
Figure 146: Sections of the Language Environment dump (Part 2 of 2)

To debug this routine, use the following steps:

1. In the dump, PLIDUMP was called by the ERROR ON-unit in statement 9. The traceback information in the dump shows that the exception occurred following statement 16.

   **Note:** In the Language Environment dumps, the columns and messages refer to "statements", but the numbers are actually (for Enterprise PL/I) the line numbers from the source file.

2. Locate the Original Condition message in the Condition Information for Active Routines section of the dump. The message is IBM0421S ONCODE=520 The SUBSCRIPTRANGE condition was raised. This message indicates that the exception occurred when an array element value exceeded the subscript range value (in this case, 10). For more information about this message, see z/OS Language Environment Runtime Messages.

3. Locate statement 14 in the routine in Figure 144 on page 303. 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 144 on page 303. 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 145 on page 304.
The block located at this offset contains the value that exceeded the array range, X'B' or 11.

**Calling a nonexistent subroutine**

Figure 147 on page 306 demonstrates the error of calling a nonexistent subroutine. This routine was compiled with the LIST, MAP, and GONUMBER compiler options. It was run with the TERMTHDACT(DUMP) runtime option to generate a traceback.

---

**Figure 147: Example of calling a nonexistent subroutine (Enterprise PL/I)**

The following examples show the traceback and condition information sections from the dump.
To understand the traceback and debug this example routine, use the following steps.

1. Find the Current Condition message in the Condition Information for Active Routines section of the dump. The message is CEE3201S The system detected an operation exception. For more information about this message, see z/OS Language Environment Runtime Messages.

   This section of the dump also provides such information as the name of the active routine and the current statement number at the time of the dump. The Location section indicates that the exception occurred at offset X’000001AE’ statement 12 within entry EXAMPLE1.

2. Locate statement 12 in the routine (Figure 147 on page 306). 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 148 on page 308 demonstrates a divide-by-zero error. In this example, the main Enterprise PL/I routine passed bad data to an Enterprise PL/I subroutine. The bad data in this example is 0, and the error occurred when the subroutine SUB1 attempted to use this data as a divisor.
Because variables are not usually displayed in a PLIDUMP dump, this routine included a PUT DATA statement, which generated a listing of arguments and variables used in the routine. Figure 149 on page 309 shows this output.
The routine in Figure 148 on page 308 was compiled with the LIST compiler option, which generated the object code listing shown in Figure 150 on page 309.

Figure 150: Object code listing from example Enterprise PL/I routine

Figure 151 on page 310 shows the Language Environment dump for routine SAMPLE.

Debugging Enterprise PL/I routines 309
Figure 151: Language Environment dump from example Enterprise PL/I routine (Part 1 of 2)
To understand the dump information and debug this routine, use the following steps:

1. Notice the title of the dump: PLIDUMP called from error ON-unit. This was the title specified when PLIDUMP was invoked, and it indicates that the ERROR condition was raised and PLIDUMP was called from within the ERROR ON-unit.

2. Locate the messages in the Condition Information section of the dump.

   There are two messages. The current condition message indicates that a prior condition was promoted to the ERROR condition. The promotion of a condition occurs when the original condition is left unhandled (no Enterprise PL/I ON-units are assigned to gain control). The original condition message is CEE3209S The system detected a fixed-point divide exception. The original condition usually indicates the actual problem. For more information about this message, see z/OS Language Environment Runtime Messages.

3. In the traceback section, note the sequence of calls in the call chain. SAMPLE called SUB1 at statement 19, and SUB1 raised an exception at statement 27, PU offset X'1C6'.

4. Find the statement in the listing for SUB1 that raised the ZERODIVIDE condition. If SUB1 was compiled with GOSTMT and SOURCE, find statement 27 in the source listing.

Since the object listing was generated in this example, you can also locate the actual assembler instruction causing the exception at offset X'1C6' in the object listing for this routine, shown in Figure 150 on page 309. Either method shows that divisor was loaded into register 2 (r2) and used as the divisor in a divide operation.
5. You can see from the declaration of SUB1 that *divisor* is a parameter passed from SAMPLE. Because of linkage conventions, you can infer that register 1 in the SAMPLE save area points to a parameter list that was passed to SUB1. *divisor* is the first parameter in the list.

6. In the SAMPLE DSA, the R1 value is X'11A3B450'. This is the address of the parameter list, which is located in static storage.

7. Find the parameter list in the stack frame; the address of the first parameter is X'11A3B484' and the value of the first parameter is X'00000000'. Thus, the exception occurred when SAMPLE passed a 0 value used as a divisor in subroutine SUB1.
Chapter 9. Debugging under CICS

This section provides information for debugging under the Customer Information Control System (CICS). The following sections explain how to access debugging information under CICS, and describe features unique to debugging under CICS.

Use the following list as a quick reference for debugging information:

- Language Environment runtime messages (CESE transient data queue)
- Language Environment traceback (CESE transient data queue)
- Language Environment dump output (CESE transient data queue)
- CICS Transaction Dump (CICS DFHDMPA or DFHDMPB data set)
- Language Environment abend and reason codes (system console)
- Language Environment return codes to CICS (system console)

If the EXEC CICS HANDLE ABEND command is active and the application, or CICS, initiates an abend or application interrupt, then Language Environment does not produce any runtime messages, tracebacks, or dumps.

If EXEC CICS ABEND NODUMP is issued, then no Language Environment dumps or CICS transaction dumps are produced.

Accessing debugging information

The following sections list the debugging information available to CICS users, and describe where you can find this information.

Under CICS, the Language Environment runtime messages, Language Environment traceback, and Language Environment dump output are written to the CESE transient data queue. The transaction identifier, terminal identifier, date, and time precede the data in the queue. For detailed information about the format of records written to the transient data queue, see z/OS Language Environment Programming Guide.

The CESE transient data queue is defined in the CICS destination control table (DCT). The CICS macro DFHDCT is used to define entries in the DCT. See CICS Resource Definition Guide for a detailed explanation of how to define a transient data queue in the DCT. If you are not sure how to define the CESE transient data queue, see your system programmer.

Locating Language Environment runtime messages

Under CICS, Language Environment runtime messages are written to the CESE transient data queue. The following example shows a Language Environment message that appears when an application abends due to an unhandled condition from an EXEC CICS command.

```
P039UTV9 19910916145313 CEE3250C The System or User ABEND AEI0 was issued. P039UTV9 19910916145313 From program unit UT9CVERI at entry point UT9CVERIT +0000011E 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 153 on page 314 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 40.

Using CICS transaction dump

The CICS transaction dump is generated to the DFHDPMA or DFHDMPB data set. The offline CICS dump utility routine converts the transaction dump into formatted, understandable output.

The CICS transaction dump contains information for the storage areas and resources associated with the current transaction. This information includes the Communication Area (COMMAREA), Transaction Work Area (TWA), Exec Interface Block (EIB), and any storage obtained by the CICS EXEC commands. This information does not appear in the Language Environment dump. It can be helpful to use the CICS transaction dump with the Language Environment dump to locate problems when operating under CICS.

When the location of an error is uncertain, it can be helpful to insert EXEC CICS DUMP statements in and around the code suspected of causing the problem. This generates CICS transaction dumps close to the error for debugging reference.

For information about interpreting CICS dumps, see CICS Problem Determination Guide.

Using CICS register and program status word contents

When a routine interrupt occurs (code = ASRA) and a CICS dump is generated, CICS formats the contents of the program status word (PSW) and the registers at the time of the interrupt. This information is also
contained in the CICS trace table entry marked SSRP * EXEC* — ABEND DETECTED. For the format of
the information contained in this trace entry, see CICS Data Areas, KERRD - KERNEL ERROR DATA.

The address of the interrupt can be found from the second word of the PSW, giving the address of the
instruction following the point of interrupt. The address of the entry point of the function can be
subtracted from this address. The offset compared to this listing gives the statement that causes the
interrupt.

For C routines, you can find the address of the entry point in register 3.

If register 15 is corrupted, the contents of the first load module of the active enclave appear in the
program storage section of the CICS transaction dump.

Using Language Environment abend and reason codes

An application can end with an abend in two ways:

- User-specified abend (that is, an abend requested by the assembler user exit or the ABTERMENC
  runtime option).
- Language Environment-detected unrecoverable error (in which case there is no Language Environment
  condition handling).

When Language Environment detects an unrecoverable error under CICS, Language Environment
terminates the transaction with an EXEC CICS abend. The abend code has a number between 4000 and
4095. A write-to-operator (WTO) is performed to write a CEE1000S message to the system console. This
message contains the abend code and its associated reason code. The WTO is performed only for
unrecoverable errors detected by Language Environment. No WTO occurs for user-requested abends.

Although this type of abend is performed only for unrecoverable error conditions, an abend code of 4000–
4095 does not necessarily indicate an internal error within Language Environment. For example, an
application routine can write a variable outside its storage and corrupt the Language Environment control
blocks.

Possible causes of a 4000–4095 abend are corrupted Language Environment control blocks and internal
Language Environment errors. For more information about abend codes 4000–4095, see z/OS Language
Environment Runtime Messages. Following is a sample Language Environment abend and reason code.
Abend codes appear in decimal, and reason codes appear in hexadecimal.

```
12.34.27 J08055B5 IEF450I XCEPII03 GO CEPII03 - 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 Runtime Messages. The
following example shows a sample of a return code that was returned to CICS.

```
+DFHAP1200I
LE03CC01 A CICS request to Language Environment has failed. Reason
code '0012030'.
```
Activating Language Environment feature trace records under CICS

Activating Language Environment feature trace records under CICS will allow users to monitor and determine the activity of a transaction. By activating the feature trace records, Level 2 trace points are added inside Language Environment at these significant points:

- Event Handle
- Set anchor
- Gives R13 and parameters before call

These trace points are useful for any support personnel that needs to know what happened inside Language Environment from a CICS call.

The function will be enabled by the existing CICS transactions. A user must enable the AP domain level 2 in order to include the Language Environment trace points. For more information on activating the CICS trace, see *CICS Diagnosis Reference*.

Every time CICS calls Language Environment, the feature trace is activated under the Extended Runtime Library Interface (ERTLI). The trace can be seen in CICS transaction dumps. Feature trace entries are formatted in a similar way to CICS trace items. There are three formats: ABBREV, SHORT & FULL. The ABBREV version (Figure 154 on page 316) just formats the heading line for each trace point and is laid out in a similar way to CICS trace entries.

![Figure 154: CICS trace output in the ABBREV format.](image-url)

The Domain Name field is replaced with a "Feature" short name (for example, Lang.Env.) and module name (for example, CEE.....) which are coded into the "Feature Trace" initialization (short name) and header formatting call (module name). See the following macro example.

The FULL version includes the heading from the ABBREV version and then dumps each captured block in Hex and Character formats. For an example, see Figure 155 on page 317.
The first block is used for the feature trace information. It contains the name of the off-line formatting module and the short name used in the formatted heading line. The other 6 blocks are available for user data.

The SHORT version is a cross between the ABBREV and FULL versions.

### Ensuring transaction rollback

If your application does not run to normal completion and there is no CICS transaction abend, take steps to ensure that transaction rollback (the backing out of any updates made by the malfunctioning application) takes place.

There are two ways to ensure that a transaction rollback occurs when an unhandled condition of severity 2 or greater is detected:

- Use the ABTERMENC runtime option with the ABEND suboption (ABTERMENC(ABEND))
- Use an assembler user exit that requests an abend for unhandled conditions of severity 2 or greater

The IBM-supplied assembler user exit for CICS (CCECXTA), 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 22 and z/OS Language Environment Programming Guide.

### Finding data when Language Environment returns a nonzero return code

Language Environment does not write any messages to the CESE transient data queue. Table 48 on page 318 shows the output generated when Language Environment returns a nonzero reason code to CICS and the location where the output appears.
Table 48: Finding data when Language Environment returns a nonzero return code

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

Finding data when Language Environment abends internally

Language Environment does not write any messages to the CESE transient data queue. Table 49 on page 318 shows the output generated when Language Environment abends internally and the location where the output appears:

Table 49: Finding data when Language Environment abends internally

<table>
<thead>
<tr>
<th>Output Message</th>
<th>Location</th>
<th>Issued By</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFHAC2206 14:35:24 LE03CC01 Transaction UTV8 has failed with abend 4095. Resource backout was successful.</td>
<td>User's terminal</td>
<td>CICS</td>
</tr>
<tr>
<td>CEE1000S LE INTERNAL abend. ABCODE = 00000FFF REASON = 00001234</td>
<td>System console</td>
<td>Language Environment</td>
</tr>
<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</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 runtime option TERMTHDACT(MSG).

Table 50: Finding data when Language Environment abends from an EXEC CICS command

<table>
<thead>
<tr>
<th>Output Message</th>
<th>Location</th>
<th>Issued By</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFHAC2206 14:35:34 LE03CC01 Transaction UTV8 has failed with abend AEI. 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</td>
<td>CICS</td>
</tr>
<tr>
<td>P021UTV9 091156 143516 CEE3250C The System or User Abend AEI0 was issued.</td>
<td>Transient data queue</td>
<td>Language Environment</td>
</tr>
</tbody>
</table>
Displaying and modifying runtime options with the CLER transaction

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

- Display the current runtime options in effect for the region.
- Modify the following subset of the region runtime options:
  - ALL31(ON|OFF)
  - CBLPSHPOP(ON|OFF)
  - CHECK(ON|OFF)
  - HEAPZONES(0-1024,QUIET|MSG|TRACE|ABEND)
  - INFOMSGFILTER(ON|OFF)
  - RPTOPTS(ON|OFF)
  - RPTSTG(ON|OFF)
  - TERMTHDACT(QUIET|MSG|TRACE|DUMP|UAONLY|UATRACE|UADUMP|UAIMM)
  - TRAP(ON|OFF)
- Write the current region runtime options to the CESE queue for printing.

The CLER transaction is conversational; it presents the user with commands for the terminal display. The runtime options that can be modified with this transaction are only in effect for the duration of the running region.

The CLER transaction must be defined in the CICS CSD (CICS System Definition file). The following definitions are required, and are in the Language Environment CEECCSD job in the SCEESAMP data set. Use the CEECCSD job to activate these definitions, or you must define them dynamically with the CICS CEDA transaction.

```
DEFINE PROGRAM(CEL4RTO) GROUP(CEE) LANGUAGE(ASSEMBLER) EXECKEY(CICS)
DEFINE MAPSET(CELCLEM) GROUP(CEE)
DEFINE MAPSET(CELCLRH) GROUP(CEE)
DEFINE TRANS(CLER) PROG(CEL4RTO) GROUP(CEE)
```

**Note:** If the runtime option ALL31 is modified to OFF, the stack is forced to BELOW. When the stack is modified to BELOW, it will remain below for the duration of the region, even if you set ALL31 back to ON. A warning message, asking if you want to continue, is presented on the panel if the runtime option ALL31 is set to OFF or CBLPSHPOP, RPTOPTS, and RPTSTG are set to ON.

To send the runtime option report to the CESE queue for output display or printing, press PF10 on the panel which displays the runtime option report.

For detailed information on the use of CLER, select PF1 from the main menu that is displayed when the CLER transaction is invoked.
Part 3. Debugging Language Environment AMODE 64 applications

This part provides specific information for debugging applications written to make use of the memory address space above the 2 GB bar.
Chapter 10. Preparing your AMODE 64 application for debugging

This chapter describes options and features that you can use to prepare your AMODE 64 application for debugging. The following topics are covered:

- Compiler options for C, C++, PL/I
- Language Environment runtime options
- Use of storage in routines
- Options for modifying exception handling
- Assembler user exits
- Enclave termination behavior
- Language Environment feedback codes and condition tokens

Setting compiler options

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

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

XL C and XL C++ compiler options for AMODE 64 applications

When compiling an application using the LP64 compiler option, you cannot use the TEST compiler option. You must instead use the DEBUG(FORMAT(DWARF)) compiler option.

When the GONUMBER compiler option is used with LP64, it will produce executables with additional debug information. This is used by Language Environment to produce statement numbers in the Language Environment dump (CEEDUMP). Statement numbers in the CEEDUMP are also produced if the DEBUG compiler option or the c89 -g option is used.

For a detailed explanation of the debugging options for XL C/C++ and Inter-procedural Analysis (IPA), see z/OS XL C/C++ User’s Guide and z/OS XL C/C++ Programming Guide.

Using Language Environment runtime options

Several runtime options affect debugging in Language Environment. The TEST runtime option, for example, can be used with a debugging tool to specify the level of control in effect for the debugging tool when the routine being initialized is started. The DYNDUMP, HEAPCHK, TERMTHDACT, TRACE, and TRAP options affect exception handling. The following Language Environment runtime options affect debugging. For a more detailed discussion of these runtime options, see z/OS Language Environment Programming Reference.

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description of runtime option</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEEDUMP</td>
<td>Specifies options to control the processing of the Language Environment dump report.</td>
</tr>
<tr>
<td>DYNDUMP</td>
<td>Provides a way to obtain IPCS readable dumps of user applications that would ordinarily be lost due to the absence of a SYSDUMP, SYSUDUMP, or SYSABEND DD statement</td>
</tr>
<tr>
<td>HEAPCHK</td>
<td>Determines whether additional heap check tests are performed.</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1991, 2018
<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description of runtime option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HEAPZONES</strong></td>
<td>Activates user heap overlay toleration and checking.</td>
</tr>
<tr>
<td><strong>INFOMSGFILTER</strong></td>
<td>Filters user specified informational messages from stderr. <strong>Note:</strong> Affects only those messages generated by Language Environment and any routine that calls <code>__le_msg_get_and_write()</code>. Other routines that write to stderr, such as <code>__le_msg_write()</code>, do not have a filtering option.</td>
</tr>
<tr>
<td><strong>PROFILE</strong></td>
<td>Controls the use of an optional profiler tool, which collects performance data for the running application. When this option is in effect, the profiler is loaded and the debugger cannot be loaded. If the TEST option is in effect when PROFILE is specified, the profiler tool will not be loaded.</td>
</tr>
<tr>
<td><strong>RPTOPTS</strong></td>
<td>Causes a report to be produced which contains the runtime options in effect. See “Determining the runtime options in effect for AMODE 64 applications” on page 324 below.</td>
</tr>
<tr>
<td><strong>RPTSTG</strong></td>
<td>Generates a report of the storage used by an enclave. See “Controlling storage allocation for AMODE 64 applications” on page 326.</td>
</tr>
<tr>
<td><strong>STORAGE</strong></td>
<td>Specifies that Language Environment initializes all heap and stack storage to a user-specified value.</td>
</tr>
<tr>
<td><strong>TERMTHDACT</strong></td>
<td>Controls response when an enclave terminates due to an unhandled condition of severity 2 or greater.</td>
</tr>
<tr>
<td><strong>TEST</strong></td>
<td>Specifies the conditions under which a debugging tool assumes control.</td>
</tr>
<tr>
<td><strong>TRACE</strong></td>
<td>Activates Language Environment runtime library tracing and controls the size of the trace table, the type of trace, and whether the trace table should be dumped unconditionally upon termination of the application.</td>
</tr>
<tr>
<td><strong>TRAP</strong></td>
<td>When TRAP is set to ON, Language Environment traps routine interrupts and abends, and optionally prints trace information or invokes a user-written exception handling routine. With TRAP set to OFF, the operating system handles all interrupts and abends. You should generally set TRAP to ON, or your runtime results can be unpredictable.</td>
</tr>
</tbody>
</table>

**Determining the runtime options in effect for AMODE 64 applications**

The runtime options in effect at the time the routine is run can affect routine behavior. Use RPTOPTS(ON) to generate an options report in the Language Environment message file when your routine terminates. The options report lists runtime options, and indicates where they were set. Figure 156 on page 325 shows a sample options report.
Understanding the HEAPZONES and HEAPCHK runtime options

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

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

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

HEAPCHK investigates the entire user heap for damage during heap related calls at a frequency based on the specified settings in the option. Because HEAPCHK will traverse the entire user heap, a slow down in application performance will occur. Information about HEAPCHK diagnostic output is discussed in Chapter 3, "Using Language Environment debugging facilities," on page 33.

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

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

### Controlling storage allocation for AMODE 64 applications

The following runtime options control storage allocation:

- **HEAP64**
- **HEAPPOOLS**
- **HEAPPOOLS64**
- **IOHEAP64**
- **LIBHEAP64**
- **STACK64**
- **THREADSTACK64**

*z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode* provides useful tips to assist with the tuning process. Appropriate tuning is necessary to avoid performance problems.

To generate a report of the storage a routine (or more specifically, an enclave) used during its run, specify the RPTSTG(ON) runtime option. The storage report, generated during enclave termination provides statistics that can help you understand how space is being consumed as the enclave runs. If storage management tuning is desired, the statistics can help you set the corresponding storage-related runtime options for future runs. Figure 157 on page 326 shows a sample storage report.

---

**Storage Report for Enclave main Tue Sep 19 09:36:22 2017**

Language Environment V02 R02.00

**STACK64 statistics:**
- Initial size: 1M
- Increment size: 1M
- Maximum used by all concurrent threads: 1M
- Largest used by any thread: 1M
- Number of increments allocated: 0

**THREADSTACK64 statistics:**
- Initial size: 1M
- Increment size: 1M
- Maximum used by all concurrent threads: 0M
- Largest used by any thread: 0M
- Number of increments allocated: 0

**64bit User HEAP statistics:**
- Initial size: 1M
- Increment size: 1M
- Total heap storage used: 983808
- Suggested initial size: 1M
- Successful Get Heap requests: 11
- Successful Free Heap requests: 0
- Number of segments allocated: 0
- Number of segments freed: 0

**31bit User HEAP statistics:**
- Initial size: 32768
- Increment size: 32768
- Total heap storage used (sugg. initial size): 243352
- Successful Get Heap requests: 58
- Successful Free Heap requests: 0
- Number of segments allocated: 9
- Number of segments freed: 0

**24bit User HEAP statistics:**
- Initial size: 4096
- Increment size: 4096
- Total heap storage used (sugg. initial size): 0
- Successful Get Heap requests: 0
- Successful Free Heap requests: 0
- Number of segments allocated: 0
- Number of segments freed: 0

**64bit Library HEAP statistics:**
- Initial size: 1M
- Increment size: 1M
- Total heap storage used: 3795584
- Suggested initial size: 4M
- Successful Get Heap requests: 384
- Successful Free Heap requests: 337
- Number of segments allocated: 2
- Number of segments freed: 0

**31bit Library HEAP statistics:**
- Initial size: 16384
- Increment size: 8192
- Total heap storage used (sugg. initial size): 0
- Successful Get Heap requests: 0
- Successful Free Heap requests: 0
- Number of segments allocated: 0
- Number of segments freed: 0

---

Figure 157: 64-bit storage report (Part 1 of 4)
Figure 159 on page 327 shows a sample storage report.

Figure 159: 64-bit storage report (Part 3 of 4)

Figure 160 on page 328 shows a sample storage report.
Storage statistics for AMODE 64 applications

The statistics for initial and incremental allocations of storage types that have a corresponding runtime option differ from the runtime option settings when their values have been rounded up by the implementation, or when allocations larger than the amounts specified were required during execution. See the descriptions of the runtime options in z/OS Language Environment Programming Reference for information about rounding.

Stack storage statistics for AMODE 64 applications

Language Environment stack storage is managed at the thread level—each thread has its own stack-type resources.

STACK64 and THREADSTACK64 statistics

- Initial size—the actual size of the initial stack area assigned to each thread. If a pthread-attributes-table is provided on the invocation of pthread-create, the stack size specified in the pthread-attributes-table takes precedence over the stack runtime options.
- Increment size—the size of each incremental stack area made available, as determined by the increment portion of the corresponding runtime option.
- Maximum used by all concurrent threads—the maximum amount allocated in total at any one time by all concurrently executing threads.
- Largest used by any thread—the largest amount allocated ever by any single thread.
- Number of increments allocated—the number of incremental segments allocated by all threads.

Determining the applicable threads

If the application is not a multithreading application, the STACK64 statistics are for the one and only thread that executed, and the THREADSTACK64 statistics are all zero.

If the application is a multithreading application, and THREADSTACK64 was not suppressed, the STACK64 statistics are for the initial thread (IPT), and the THREADSTACK64 statistics are for the other threads. However, if THREADSTACK64 was suppressed, the STACK64 statistics are for all of the threads, initial and other.

Allocating stack storage

The allocation of the stack for each thread, including the initial processing thread (IPT), is part of a storage request to the system when the thread is first created. Other storage, not part of the stack, is also acquired at this time. These storage allocations are not shown in the storage report. The size of the stack portion of this storage is the stack maximum size plus a one megabyte (1M) guard area. After allocation, the guard area follows the stack initial size and runs through the end of the stack maximum size plus the 1M guard area. Increments to the stack for each thread do not result in additional storage requests to the system. They result in the movement of the beginning of the guard area no further than the maximum size.
of the stack. The stack initial, increment, and maximum sizes are controlled through the STACK64 and THREADSTACK64 runtime options.

**Heap storage statistics**

Language Environment heap storage is managed at the enclave level. Each enclave has its own heap type resources, which are shared by the threads that execute within the enclave. The heap resources have 64-bit, 31-bit, and 24-bit addressable areas, each of which can be tuned separately.

**HEAP64, LIBHEAP64, and IOHEAP64 statistics**

- Initial size—the default initial allocation, as specified by the corresponding runtime option.
- Increment size—the minimum incremental allocation, as specified by the corresponding runtime option.
- Total heap storage used—the largest total amount used by the enclave at any one time.
- Successful Get Heap requests—the number of get heap requests.
- Successful Free Heap requests—the number of free heap requests.
- Number of segments allocated—the number of incremental segments allocated.
- Number of segments freed—the number of incremental segments individually freed.

The number of Free Heap requests could be less than the number of Get Heap requests if the pieces of heap storage acquired by individual Get Heap requests were not explicitly freed, but were freed implicitly during enclave termination. The number of incremental segments individually freed could be less than the number allocated if the segments were not explicitly freed, but were freed implicitly during enclave termination. The initial segment is included in Number of segments allocated for each 31-bit and 24-bit addressable heap resource, and for the 64-bit addressable IOHEAP64 resource. A disposition of KEEP always causes 0 to be reported for the Number of segments freed. These statistics, in all cases, specify totals for the entire enclave.

**Heap pools storage statistics**

The HEAPPOOLS and HEAPPOOLS64 runtime options for C/C++ applications only controls usage of the heap pools storage algorithm at the enclave level. The heap pools algorithm allows for the definition of one to twelve heap pools, each consisting of a number of storage cells of a specified length. For further details regarding heap pools storage statistics in the storage report, see “Language Environment storage report with heap pools statistics” on page 463.

**Modifying exception handling behavior**

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

- Application program interfaces (API)
- User-written exception handlers
- POSIX functions (used to specifically set signal actions and signal masks)

**Language Environment application program interfaces (API)**

You can use the following APIs to modify exception handling:

<table>
<thead>
<tr>
<th>Function name</th>
<th>API description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cabend()</td>
<td>Terminates an enclave using an abend.</td>
</tr>
<tr>
<td>__le_cib_get()</td>
<td>Returns a pointer to a condition information block (CIB) associated with a given condition token. The CIB contains detailed information about the condition.</td>
</tr>
<tr>
<td>__set_exception_handler()</td>
<td>Activates a routine to handle an exception.</td>
</tr>
<tr>
<td>__reset_exception_handler()</td>
<td>Removes handling of an exception by any routine.</td>
</tr>
</tbody>
</table>
Language Environment runtime options

The following Language Environment runtime options can affect your routine’s exception handling behavior:

<table>
<thead>
<tr>
<th>Runtime option</th>
<th>Description of runtime option</th>
</tr>
</thead>
</table>
| **TERMTHDACT** | Sets the level of information that is produced when a condition of severity 2 or greater remains unhandled within the enclave. The possible parameter settings for different levels of information are:  
  • QUIET for no information  
  • MSG for message only  
  • TRACE for message and a traceback  
  • DUMP for message, traceback, and Language Environment dump  
  • UAONLY for message and a system dump of the user address space  
  • UATRACE for message, Language Environment dump with traceback information only, and a system dump of the user address space  
  • UADUMP for message, traceback, Language Environment dump, and system dump  
  • UAIMM for a system dump of the user address space of the original abend or program interrupt prior to the Language Environment condition manager processing the condition. |
| **TRAP(ON)** | Fully enables the Language Environment exception handler. This causes the Language Environment exception handler to intercept error conditions and routine interrupts. When TRAP(ON, NOSPIE) is specified, Language Environment handles all program interrupts and abends through an ESTAE. Use this feature when you do not want Language Environment to issue an ESPIE macro. During normal operation, you should use TRAP(ON) when running your applications. |
| **TRAP(OFF)** | Disables the Language Environment condition handler from handling abends and program checks/interrupts. ESPIE is not issued with TRAP(OFF). Specify TRAP(OFF) when you do not want Language Environment to issue an ESPIE. When TRAP(OFF), TRAP(OFF, SPIE), or TRAP(OFF, NOSPIE) is specified and either a program interrupt or abend occurs, the user exit for termination is ignored. TRAP(OFF) can cause several unexpected side effects. It is not supported in AMODE 64 production execution. For further information, see the TRAP runtime option in z/OS Language Environment Programming Reference. |

Customizing exception handlers

User-written exception handlers permit you to customize exception handling for certain conditions. You can register a user-written exception handler for the current stack frame by using the __set_exception_handler() API. For more information about user-written exception handlers and the Language Environment condition manager, see z/OS XL C/C++ Programming Guide.

Using condition information

If a condition that might require attention occurs while an application is running, Language Environment builds a condition token. The condition token contains 16 bytes (128 bits) of information about the condition that Language Environment or your routines can use to respond appropriately. Each condition is associated with a single Language Environment runtime message. You can use this condition information in two ways:

- To specify the feedback code parameter when calling Language Environment services (see “Using the feedback code parameter” on page 331).
- To code a symbolic feedback code in a user-written exception handler (see “Using the symbolic feedback code” on page 332).

### Using the feedback code parameter

The feedback code is an optional parameter of the Language Environment APIs. For C/C++ applications, this parameter is optional. For more information about feedback codes and condition tokens, see *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*.

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

The condition token does not apply to asynchronous signals. For a discussion of the distinctions between synchronous signals and asynchronous signals with POSIX(ON), see *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*.

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

Language Environment provides APIs that can be used to convert condition tokens to routine variables, messages, or signaled conditions. The following table lists these Language Environment APIs and their functions. For more information on these APIs, see *z/OS XL C/C++ Programming Guide*.

<table>
<thead>
<tr>
<th>API name</th>
<th>API description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__le_msg_write()</td>
<td>Writes a message string to stderr</td>
</tr>
<tr>
<td>__le_msg_get_and_write()</td>
<td>Takes a message associated with a condition and writes it to stderr</td>
</tr>
<tr>
<td>__le_msg_get()</td>
<td>Retrieves, formats, and stores message data for a condition</td>
</tr>
<tr>
<td>__le_msg_add_insert()</td>
<td>Creates a message insert</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Case 1 Condition Token</th>
<th>Case 2 Condition Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-31</td>
<td>Condition_ID</td>
<td>Condition_ID</td>
</tr>
<tr>
<td>32-33</td>
<td>Case Number</td>
<td>Class</td>
</tr>
<tr>
<td>34-36</td>
<td>Severity Number</td>
<td>Cause Code</td>
</tr>
<tr>
<td>37-39</td>
<td>Control Code</td>
<td>IS1</td>
</tr>
<tr>
<td>40-63</td>
<td>Facility_ID</td>
<td></td>
</tr>
</tbody>
</table>

For Case 1 condition tokens, **Condition_ID** is:

```
0 - 15  Severity Number
16 - 31 Message Number
```

For Case 2 condition tokens, **Condition_ID** is:

```
0 - 15  Class
16 - 31 Cause Code
```

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

*Figure 161: Language Environment condition token*

For example, in the condition token: X'0003032D 59C3C5C5 00000000 00000000'
- X'0003' is severity.
- X'032D' is message number 813.
• X'59' are hexadecimal flags for case, severity, and control.
• X'C3C5C5' is the CEE facility ID.
• X'00000000 00000000' is the instance specific information (ISI). (In this case, no ISI was provided.)

If a Language Environment traceback or dump is generated while a condition token is being processed or when a condition exists, Language Environment writes the runtime message to the condition section of the traceback or dump. If a condition is detected when a Language Environment API is invoked without a feedback code, the condition token is passed to the Language Environment condition manager. If a condition is severity 0 or 1, Language Environment resumes without issuing a message. For conditions of severity 2 or greater, Language Environment issues a message and terminates. For a list of Language Environment runtime messages and corrective information, see z/OS Language Environment Runtime Messages.

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

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

Using the symbolic feedback code

The symbolic feedback code represents the first 8 bytes of a 16-byte condition token. You can think of the symbolic feedback code as the nickname for a condition. As such, the symbolic feedback code can be used in user-written exception handlers to screen for a given condition, even if it occurs at different locations in an application. For more details on symbolic feedback codes, see z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode.
Chapter 11. Classifying AMODE 64 application errors

This chapter describes errors that commonly occur in Language Environment AMODE 64 applications. It also explains how to use runtime messages and abend codes to obtain information about errors in your application.

Identifying problems in routines

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

Language Environment module names

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

- CEE (Language Environment)
- CEL (Language Environment)
- EDC (C/C++)

Module elements or text files with other prefixes are not part of the Language Environment product for AMODE 64 applications.

Common errors in routines

These common errors have simple solutions:

- If you receive abend U4093, reason X'224' (548 decimal), then make sure you use MEMLIMIT to allow access to above the 2 GB bar. For more information, see z/OS MVS Programming: Extended Addressability Guide.
- If you do not have enough virtual storage, increase your region size or decrease your storage usage (stack size) by using the storage-related runtime options and callable services. (See “Controlling storage allocation for AMODE 64 applications” on page 326 for information about using storage in routines.)
- If you do not have enough disk space, increase your disk allocation.
- If executable files are not available, check your executable library to ensure that they are defined. For example, check your STEPLIB or JOBLIB definitions.

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

Changes in optimization levels, addressing modes, and input/output file formats can also cause unanticipated problems in your routine.

In most cases, generated condition tokens or runtime messages point to the nature of the error. The runtime messages offer the most efficient corrective action. To help you analyze errors and determine the most useful method to fix the problem, Table 51 on page 334 lists common error symptoms, possible causes, and programmer responses.
<table>
<thead>
<tr>
<th>Error Symptom</th>
<th>Possible Cause</th>
<th>Programmer Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbered runtime message appears</td>
<td>Condition raised in routine</td>
<td>For any messages you receive, read the Programmer Response. For information about</td>
</tr>
<tr>
<td></td>
<td></td>
<td>message structure, see “Interpreting runtime messages” on page 334 below.</td>
</tr>
<tr>
<td>User abend code &lt; 4000</td>
<td>• A non-Language Environment abend occurred</td>
<td>See the Language Environment abend codes in z/OS Language Environment Runtime</td>
</tr>
<tr>
<td></td>
<td>• The assembler user exit requested an abend for an unhandled condition of</td>
<td>Messages. Check for a subsystem-generated abend or a user-specified abend.</td>
</tr>
<tr>
<td></td>
<td>severity ≥2</td>
<td></td>
</tr>
<tr>
<td>User abend code ≥ 4000</td>
<td>• Language Environment detected an error and could not proceed</td>
<td>For any abends you receive, read the appropriate explanation listed in the abend</td>
</tr>
<tr>
<td></td>
<td>• An unhandled software-raised condition occurred</td>
<td>codes section of z/OS Language Environment Runtime Messages.</td>
</tr>
<tr>
<td></td>
<td>• The assembler user exit requested an abend for an unhandled condition of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>severity 4</td>
<td></td>
</tr>
<tr>
<td>System abend with TRAP(OFF)</td>
<td>Cause depends on type of malfunction</td>
<td>Respond appropriately. See the messages and codes book of the operating system.</td>
</tr>
<tr>
<td>System abend with TRAP(ON)</td>
<td>System-detected error</td>
<td>See the messages and codes book of the operating system.</td>
</tr>
<tr>
<td>No response (wait/loop)</td>
<td>Application logic failure</td>
<td>Check routine logic.</td>
</tr>
<tr>
<td>Unexpected message (message received was</td>
<td>Condition caused by something related to current service</td>
<td>Generate a traceback using cdump() or ctrace().</td>
</tr>
<tr>
<td>not from most recent service)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incorrect output</td>
<td>Incorrect file definitions, storage overlay, incorrect routine mask setting,</td>
<td>Correct the appropriate parameters.</td>
</tr>
<tr>
<td></td>
<td>references to uninitialized variables, data input errors, or application</td>
<td></td>
</tr>
<tr>
<td></td>
<td>routine logic error</td>
<td></td>
</tr>
<tr>
<td>No output</td>
<td>Incorrect ddname or file definitions</td>
<td>Correct the appropriate parameters.</td>
</tr>
<tr>
<td>Nonzero return code from enclave</td>
<td>The return code was issued by the application routine</td>
<td>Check the application for the meaning of the return code.</td>
</tr>
</tbody>
</table>

**Interpreting runtime messages**

The first step in debugging your routine is to look up any runtime messages. Runtime messages are written to the C stderr stream. Runtime messages provide users with additional information about a condition, and possible solutions for any errors that occurred. They can be issued by Language Environment common routines or language-specific runtime routines and contain a message prefix, message number, severity code, and descriptive text.
In the following example Language Environment message:

```
CEE3206S The system detected a specification exception (System Completion Code=OC6).
```

- 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++ runtime library routines commonly use the Language Environment services. This is why you can see Language Environment messages even when the application routine does not directly call common runtime services.

**Message prefix**

The message prefix indicates the Language Environment component that generated the message. The message prefix is the first three characters of the message number and is also the facility ID in the condition token. The messages for the various components can be found in z/OS Language Environment Runtime Messages.

<table>
<thead>
<tr>
<th>Message Prefix</th>
<th>Language Environment Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEE</td>
<td>Common run time</td>
</tr>
<tr>
<td>EDC</td>
<td>C/C++ run time</td>
</tr>
</tbody>
</table>

**Message number**

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

**Severity code**

The severity code is the letter following the message number and indicates the level of attention called for by the condition. Messages with severity "I" are informational messages and do not usually require any corrective action. In general, if more than one runtime message appears, the first noninformational message indicates the problem. For a complete list of severity codes, severity values, condition information, and default actions, see z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode.

**Message text**

The message text provides a brief explanation of the condition.

**Understanding abend codes**

Under Language Environment, abnormal terminations generate abend codes. There are two types of abend codes: 1) user abends (Language Environment and user-specified) and 2) system abends. User abends follow the format of Udddd, where dddd is a decimal user abend code. System abends follow the format of Shhh, where 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: S88A

Classifying AMODE 64 application errors 335
The Language Environment API `__cabend()` terminates your application with an abend. You can set the `clean_up` parameter value to determine how the abend is processed and how Language Environment handles the raised condition. For more information about `__cabend()` and `clean_up`, see z/OS XL C/C++ Runtime Library Reference.

**User abends**

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

**System abends**

If you receive a system abend code, look up the code and the corresponding information in the publications for the system you are using. When a system abend occurs, the operating system can generate a system dump. System dumps are written to ddname SYSMDUMP, SYSABEND, or SYSUDUMP. If the DYNDUMP runtime option is used in combination with the TERMTHDACT runtime option, the system dump can be written without the ddname specified. System dumps show the memory state at the time of the condition. See “Generating a system dump” on page 354 for more information about system dumps.
Chapter 12. Using Language Environment AMODE 64 debugging facilities

This section describes methods of debugging AMODE 64 routines in Language Environment. Currently, most problems in Language Environment and member language routines can be determined through the use of a debugging tool or through information provided in the Language Environment dump.

Debugging tools

You can use `dbx` to debug Language Environment applications. *z/OS UNIX System Services Command Reference* has information on `dbx` subcommands, while *z/OS UNIX System Services Programming Tools* contains usage information.

Language Environment dumps

The following sections provide information about using the Language Environment dump service, and describe the contents of the Language Environment dump.

Generating a Language Environment dump with TERMTHDACT

The TERMTHDACT runtime option produces a dump during program checks or abnormal terminations. You must use TERMTHDACT(DUMP) in conjunction with TRAP(ON) to generate a Language Environment dump. You can use TERMTHDACT to produce a traceback, Language Environment dump, or user address space dump when a thread ends abnormally because of an unhandled condition of severity 2 or greater. If this is the last thread in the process, the enclave goes away. A thread terminating in a non-POSIX environment is analogous to an enclave terminating. For information on enclave termination, see *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*.

The TERMTHDACT suboptions QUIET, MSG, TRACE, DUMP, UAONLY, UATRACE, UADUMP, and UAIMM control the level of information available. Following are the suboptions, the levels of information produced, and the destination of each.

<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIET</td>
<td>No information</td>
<td>No destination.</td>
</tr>
<tr>
<td>MSG</td>
<td>Message</td>
<td>Stderr</td>
</tr>
<tr>
<td>TRACE</td>
<td>Message and Language Environment dump containing only a traceback</td>
<td>Message goes to stderr. Traceback goes to CEEDUMP file.</td>
</tr>
<tr>
<td>DUMP</td>
<td>Message and complete Language Environment dump</td>
<td>Message goes to stderr. Language Environment dump goes to CEEDUMP file.</td>
</tr>
<tr>
<td>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.</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>
</tbody>
</table>

Note: A Language Environment dump is not generated.
<table>
<thead>
<tr>
<th>Suboption</th>
<th>Level of information</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UATRACE</strong></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 CEE_DUMP 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><strong>UADUMP</strong></td>
<td>Message, Language Environment dump, and SYMDUMP, 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 CEE_DUMP 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><strong>UAIMM</strong></td>
<td>Language Environment generates a system dump of the original abend/program interrupt of the user address space. You will get a system dump of your user address space if the appropriate DD statement is used. After the dump is taken by the operating system, Language Environment condition manager continues processing.</td>
<td>Message goes to stderr. User address space dump goes to ddname specified for z/OS.</td>
</tr>
</tbody>
</table>

The TRACE and UATRACE suboptions of TERMTHDACT use these dump options:
- CONDITION
- ENCLAVE(ALL)
- FILES
- FNAME(CEEDUMP)
- GENOPTS
- NOBLOCKS
- NOENTRY
- NOSTORAGE
- STACKFRAME(ALL)
- THREAD(ALL)
- TRACEBACK
- VARIABLES

The DUMP and UADUMP suboptions of TERMTHDACT use these dump options:
- BLOCKS
- CONDITION
- ENCLAVE(ALL)
- FILES
- FNAME(CEEDUMP)
- GENOPTS
- NOENTRY
- STACKFRAME(ALL)
- STORAGE
- THREAD(ALL)
- TRACEBACK
- VARIABLES
Considerations for setting TERMTHDACT options

Review the following considerations before setting TERMTHDACT runtime options. For more information, see z/OS Language Environment Programming Reference.

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

• Preinitialized Environments for Authorized Programs Considerations
  - The TERMTHDACT suboptions TRACE, DUMP, UADUMP, UATRACE are overridden to UAONLY.
  - For UAONLY, a U4039 abend is generated and an SVC dump of the U4039 abend with the following title is taken:

```
COMPON=CEL,COMPID=568819801,ISSUER=CELAFRR ,MODULE=CELAEICT+????, ABEND=U4039,REASON=00000000
```

  - For UAIMM, an SVC dump of the original abend/program interrupt with the following title is taken (the ABEND and REASON values are those of the original abend/program interrupt):

```
COMPON=CEL,COMPID=568819801,ISSUER=CELAFRR ,MODULE=CELAEICT+????, ABEND=S00C9,REASON=00000009
```

Generating a Language Environment dump with language-specific functions

C/C++ routines can use the functions cdump(), csnap(), and ctrace() to produce a Language Environment dump. For more information on these functions, see “Generating a Language Environment dump of a C/C++ routine” on page 440.

Understanding the Language Environment dump

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

Figure 165 on page 342 illustrates a dump for enclave main. The example shows full use of the TERMTHDACT dump options. Ellipses are used to summarize some sections of the dump and information regarding unhandled conditions may not be present at all. Sections of the dump are numbered to correspond with the descriptions given in Figure 162 on page 340.

The CEE3DMP was generated by the C program CELQSAMP shown in Figure 162 on page 340. CELQSAMP uses the DLL CELQDLL shown in Figure 164 on page 341.
Figure 162: The C program CELQSAMP (AMODE 64) (Part 1 of 2)

The second part of the C program CELQSAMP is shown in Figure 163 on page 341.
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 163: The C program CELQSAMP (AMODE 64) (Part 2 of 2)

The DLL CELQDLL is shown in Figure 164 on page 341.

/* DLL containing div_zero */
#pragma options(SERVICE("1.4.f.0001"),NOOPT,GONUM)
#pragma export(div_zero)
#include <stdio.h>
#include <stdlib.h>
/***************************************************************************/
/*  div_zero: Cause divide by zero exception                             */
/***************************************************************************/
void *div_zero(void *parm)
{
    int                  i = 0;
    printf("Divide by zero...\n");
    i = 1/i;
    printf("Error -- Should not get here. i=%d\n",i);
    exit(110);
}

Figure 164: The C DLL CELQDLL (AMODE 64)

For easy reference, the sections of the following dump are numbered to correspond with the descriptions in “Sections of the Language Environment dump” on page 350.
Condition processing resulted in the unhandled condition.

Condition Information for PLPSC://'POSIX.CRTL.C(CELQDLL)' (DSA address 0000001082F2F008)

CIB Address: 0000001082F2F008

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

Location: 
Program Unit: PLPSC://'POSIX.CRTL.C(CELQDLL)' 
Entry: div_zero Statement: +0000004E Machine State: 
ILC..... 0002    Interruption Code..... 0009
PSW..... 0785240180000000 000000002575B5F0
GPR0..... 0000000000000000  GPR1..... 0000000100009DF0  GPR2..... 00000001082FF278  GPR3..... 0000000000000012
GPR4..... 00000001082FF080  GPR5..... 00000000000000C0  GPR6..... 0000000000000000  GPR7..... 0000000000000001
GPR8..... 000000002575B5AC  GPR9..... 000000002575B638  GPR10.... 00000000250014B0  GPR11.... 0000000108FC5E70
GPR12.... 0000000100005340  GPR13.... 0000000000006F58  GPR14.... 0000000025250098  GPR15.... 000000000000001F

[6] Condition Information for Active Routines

Condition Information for PLPSC://'POSIX.CRTL.C(CELQDLL)' (DSA address 0000001082F2F008)

CIB Address: 00000001082F2F008

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

Location: 
Program Unit: PLPSC://'POSIX.CRTL.C(CELQDLL)' 
Entry: div_zero Statement: +0000004E Machine State: 
ILC..... 0002    Interruption Code..... 0009
PSW..... 0785240180000000 000000002575B5F0
GPR0..... 0000000000000000  GPR1..... 0000000100009DF0  GPR2..... 00000001082FF278  GPR3..... 0000000000000012
GPR4..... 00000001082F2F008  GPR5..... 00000000000000C0  GPR6..... 0000000000000000  GPR7..... 0000000000000001
GPR8..... 000000002575B5AC  GPR9..... 000000002575B638  GPR10.... 00000000250014B0  GPR11.... 0000000108FC5E70
GPR12.... 0000000100005340  GPR13.... 0000000000006F58  GPR14.... 0000000025250098  GPR15.... 000000000000001F

Figure 165: Example dump using CEE3DMP (AMODE 64) (Part 1 of 9)
The following is the second part of the example dump using CEE3DMP (AMODE 64).

Storage dump near condition, beginning at location(00000002575B5DE).
  ...x....| ...&.....|  |..T.....|xx....| -..|
  +0000 00000002575B5DE  0700E300 48C00014 47000081 8E400020 |...|...|...
  +0010 00000002575B5E  1D60B904 00075000 48C0E320 48C00014 |......|...
GPREG STORAGE:
  Storage around GPR0 (0000000000000000):   Inaccessible storage.
  Storage around GPR1 (0000000000000000):   Inaccessible storage.
  Storage around GPR2 (0000000000000000):   Inaccessible storage.
  Storage around GPR3 (0000000000000000):   Inaccessible storage.
  Storage around GPR4 (00000001082FF080):   Inaccessible storage.
  Storage around GPR5 (00000001082FF180):   Inaccessible storage.
  Storage around GPR6 (00000001082FF280):   Inaccessible storage.
  Storage around GPR7 (00000001082FF380):   Inaccessible storage.
  Storage around GPR8 (00000001082FF480):   Inaccessible storage.
  Storage around GPR9 (00000001082FF580):   Inaccessible storage.
  Storage around GPR10(00000001082FF680):   Inaccessible storage.
  Storage around GPR11(00000001082FF780):   Inaccessible storage.
  Storage around GPR12(00000001082FF880):   Inaccessible storage.
  Storage around GPR13(00000001082FF980):   Inaccessible storage.
  Storage around GPR14(00000001082FFA80):   Inaccessible storage.
  Storage around GPR15(00000001082FFB80):   Inaccessible storage.

Parameters, Registers, and Variables for Active Routines:
  div_zero (DSA address 00000001082FF800):
    DOWNSTACK DSA
    Saved Registers:
    GPR0..... ****************  GPR1..... ****************  GPR2..... ****************  GPR3..... ****************
    GPR4..... 00000001082FF180  GPR5..... 00000001083710A0  GPR6..... 000000002575B5A0  GPR7..... 0000000025000542
    GPR8..... 00000000250000E4  GPR9..... 0000000025000658  GPR10.... ****************  GPR11.... ****************
    GPR12.... ****************  GPR13.... ****************  GPR14.... ****************  GPR15.... ****************
    GPREG STORAGE:
    Storage around GPR0 is invalid.
    Storage around GPR1 is invalid.
    Storage around GPR2 is invalid.
    Storage around GPR3 is invalid.
    Storage around GPR4 is invalid.
    Storage around GPR5 is invalid.
    Storage around GPR6 is invalid.
    Storage around GPR7 is invalid.
    Storage around GPR8 is invalid.
    Storage around GPR9 is invalid.
    Storage around GPR10 is invalid.
    Storage around GPR11 is invalid.
    Storage around GPR12 is invalid.
    Storage around GPR13 is invalid.
    Storage around GPR14 is invalid.
    Storage around GPR15 is invalid.
  CELQINIT (DSA address 00000001082FF280):
    DOWNSTACK DSA
    Saved Registers:
    GPR0..... ****************  GPR1..... ****************  GPR2..... ****************  GPR3..... ****************
    GPR4..... 00000001082FF180  GPR5..... 00000001083710A0  GPR6..... 000000002575B5A0  GPR7..... 0000000025000542
    GPR8..... 00000000250000E4  GPR9..... 0000000025000658  GPR10.... ****************  GPR11.... ****************
    GPR12.... ****************  GPR13.... ****************  GPR14.... ****************  GPR15.... ****************
    GPREG STORAGE:
    Storage around GPR0 is invalid.
    Storage around GPR1 is invalid.
    Storage around GPR2 is invalid.
    Storage around GPR3 is invalid.
    Storage around GPR4 is invalid.
    Storage around GPR5 is invalid.
    Storage around GPR6 is invalid.
    Storage around GPR7 is invalid.
    Storage around GPR8 is invalid.
    Storage around GPR9 is invalid.
    Storage around GPR10 is invalid.
    Storage around GPR11 is invalid.
    Storage around GPR12 is invalid.
    Storage around GPR13 is invalid.
    Storage around GPR14 is invalid.
    Storage around GPR15 is invalid.

Figure 166: Example dump using CEE3DMP (AMODE 64) (Part 2 of 9)
Figure 167: Example dump using CEE3DMP (AMODE 64) (Part 3 of 9)
The following is the fourth part of the example dump using CEE3DMP (AMODE 64).

<table>
<thead>
<tr>
<th>Address</th>
<th>Storage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0060</td>
<td>00000001082FBE60</td>
<td>00000000 25180000 00000000 00000000</td>
</tr>
<tr>
<td>+0070</td>
<td>00000001082FBE70</td>
<td>00000001 082F0800 00000000 257880F0</td>
</tr>
<tr>
<td>+0090</td>
<td>00000001082FBE90</td>
<td>00000001 082F0800 01000000 00000000</td>
</tr>
<tr>
<td>+00A0</td>
<td>00000001082FBEA0</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+00B0</td>
<td>00000001082FBB0</td>
<td>+0000FF 000000300000FF</td>
</tr>
<tr>
<td>+00C0</td>
<td>00000001082FBC0</td>
<td>00000000 940C0000 00000009</td>
</tr>
<tr>
<td>+00D0</td>
<td>00000001082FBD0</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+00E0</td>
<td>00000001082FBE0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+00F0</td>
<td>00000001082FBBF0</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>+0100</td>
<td>00000001082FBCF0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0110</td>
<td>00000001082FBDF0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0120</td>
<td>00000001082FCE0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0130</td>
<td>00000001082FDF0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0140</td>
<td>00000001082FE0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0150</td>
<td>00000001082FE10</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0160</td>
<td>00000001082FE20</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0170</td>
<td>00000001082FE30</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0180</td>
<td>00000001082FE40</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0190</td>
<td>00000001082FE50</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+01A0</td>
<td>00000001082FE60</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+01B0</td>
<td>00000001082FE70</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+01C0</td>
<td>00000001082FE80</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+01D0</td>
<td>00000001082FE90</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+01E0</td>
<td>00000001082FEA0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+01F0</td>
<td>00000001082FEB0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0200</td>
<td>00000001082FEC0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0210</td>
<td>00000001082FED0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0220</td>
<td>00000001082FEE0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0230</td>
<td>00000001082FEF0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0240</td>
<td>00000001082FF00</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0250</td>
<td>00000001082FF10</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0260</td>
<td>00000001082FF20</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0270</td>
<td>00000001082FF30</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0280</td>
<td>00000001082FF40</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0290</td>
<td>00000001082FF50</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+02A0</td>
<td>00000001082FF60</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+02B0</td>
<td>00000001082FF70</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+02C0</td>
<td>00000001082FF80</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+02D0</td>
<td>00000001082FF90</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+02E0</td>
<td>00000001082FFA0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+02F0</td>
<td>00000001082FFB0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0300</td>
<td>00000001082FFC0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0310</td>
<td>00000001082FFD0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0320</td>
<td>00000001082FFE0</td>
<td>00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0330</td>
<td>00000001082FFF0</td>
<td>00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

DSA for main: 00000001082FDFE80

---

Figure 168: Example dump using CEE3DMP (AMODE 64) (Part 4 of 9)
The following is the fifth part of the example dump using CEE3DMP (AMODE 64).

Figure 169: Example dump using CEE3DMP (AMODE 64) (Part 5 of 9)
The following is the sixth part of the example dump using CEE3DMP (AMODE 64).

[10] Control Blocks Associated with the Thread:
CXA(00000001114013C8)
+000 00000001114013C8 00000000 00000000 00000000 00000000 |................|
+001 00000001114013D8 +0001AF 0000000111401677 same as above
+02B 0000000111401678 00008000 00000000 00000000 00000000 |................|

[11] Enclave Control Blocks:
EDB(0000000100005340)
+000 0000000100005340 C3C5C5C5 C4C24040 00000000 00000000 |CEEEDB ........|
+001 0000000100005350 00000000 00000000 00000000 00000000 |................|
+002 0000000100005360 +0000FF 000000010000543F same as above
+010 0000000100005440 97000100 00000001 00068F8 |p..............8|

Mutex and Condition Variable Blocks (MCVB+MHT+CHT)(00000001089100B8)
+000 00000001089100B8 00000000 00011E78 00000001 08910100 |.............j..|
+001 00000001089100C8 00007F00 00007F00 00000000 00000000 |...0............|
+002 00000001089100D8 00000001 08FC7470 00000001 08910900 |.............j..|
+003 00000001089100E8 000001F0 00001F00 00000000 00000000 |...0............|
+

Thread Synchronization Enclave Latch Table (EPALT)(0000000108910B00)
+000 0000000108910B00 00000000 00000000 00000000 00000000 |................|
+001 0000000108910B10 +00015F 0000000108910C5F same as above
+016 0000000108910C60 00000000 DA8ADF60 00000000 |...........-....|
+017 0000000108910C70 00000000 257520A0 00000118 |.............j..|
+018 0000000108910C80 00000000 00000000 00000000 |................|
+019 0000000108910C90 +00022F 0000000108910DC7 same as above
+

Heaps:
HEAPCHK Option Control Block (HCOP)(00000001089124D0)
+000 00000001089124D0 00000000 00000000 00000000 00000000 |HCOP............|
+001 00000001089124E0 00000000 00000000 00000000 00000000 |................|
+002 00000001089124F0 00000000 00000000 00000000 00000000 |................|
+003 0000000108912500 00000000 00000000 00000000 00000000 |................|
+004 0000000108912510 00000000 00000000 00000000 00000000 |................|

HEAPCHK Element Table (HECEL) for Heapid 00000001089124D0:
Address Seg Address Length
Figure 170: Example dump using CEE3DMP (AMODE 64) (Part 6 of 9)
The following is the seventh part of the example dump using CEE3DMP (AMODE 64).

Figure 171: Example dump using CEE3DMP (AMODE 64) (Part 7 of 9)
Table 172: Example dump using CEE3DMP (AMODE 64)

<table>
<thead>
<tr>
<th>Time</th>
<th>Date</th>
<th>Thread ID</th>
<th>Entry Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.19.55.718016</td>
<td>2007.01.17</td>
<td>253E019000000000</td>
<td></td>
</tr>
<tr>
<td>21.19.55.719149</td>
<td>2007.01.17</td>
<td>253E1FB0000000002</td>
<td></td>
</tr>
<tr>
<td>21.19.55.719713</td>
<td>2007.01.17</td>
<td>253E10A0000000001</td>
<td></td>
</tr>
<tr>
<td>21.19.55.719939</td>
<td>2007.01.17</td>
<td>253E1FB0000000002</td>
<td></td>
</tr>
<tr>
<td>21.19.55.719935</td>
<td>2007.01.17</td>
<td>253E019000000000</td>
<td></td>
</tr>
<tr>
<td>21.19.55.719939</td>
<td>2007.01.17</td>
<td>253E10A0000000000</td>
<td></td>
</tr>
</tbody>
</table>

Heap Storage Diagnostics

All storage has been freed.

Figure 172: Example dump using CEE3DMP (AMODE 64) (Part 8 of 9)
The following is the ninth part of the example dump using CEE3DMP (AMODE 64).

**Sections of the Language Environment dump**

The sections of the dump listed in Table 53 on page 350 appear independently of the Language Environment-conforming languages used.

Table 53: Contents of the Language Environment dump - AMODE 64.

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Page Heading</td>
<td>The page heading section appears on the top of each page of the dump and contains:</td>
</tr>
<tr>
<td></td>
<td>• CEE3DMP identifier</td>
</tr>
<tr>
<td></td>
<td>• Title For dumps generated as a result of an unhandled condition, the title is &quot;Condition processing resulted in the Unhandled condition.&quot;</td>
</tr>
<tr>
<td></td>
<td>• Product abbreviation of Language Environment</td>
</tr>
<tr>
<td></td>
<td>• Version number</td>
</tr>
<tr>
<td></td>
<td>• Release number</td>
</tr>
<tr>
<td></td>
<td>• Date</td>
</tr>
<tr>
<td></td>
<td>• Time</td>
</tr>
<tr>
<td></td>
<td>• Page number</td>
</tr>
<tr>
<td>[2] CEE3845I CEEPUMP Processing started.</td>
<td>Identifies the start of the Language Environment dump processing. Similarly, message CEE3846I identifies the end of the dump processing, Message number CEE3845I can be used to locate the start of the next CEEPUMP report when scanning forward in a data set that contains several CEEPUMP reports.</td>
</tr>
<tr>
<td>[3] Enclave Identifier</td>
<td>Names the enclave for which information in the dump is provided.</td>
</tr>
</tbody>
</table>
Table 53: Contents of the Language Environment dump - AMODE 64. (continued)

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[4] - [10] Thread Information:</td>
<td>These sections show information that is specific to a thread. When multiple threads are dumped, these sections will appear for each thread.</td>
</tr>
<tr>
<td>[4] Information for thread</td>
<td>Shows the system identifier for the thread. Each thread has a unique identifier.</td>
</tr>
<tr>
<td>[5] Traceback</td>
<td>For all active routines in a particular thread, the traceback section shows routine information in three parts. The first part contains:</td>
</tr>
<tr>
<td></td>
<td>• DSA number: A number that is assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback.</td>
</tr>
<tr>
<td></td>
<td>• Entry: For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, then the string '* NoName **' will appear.</td>
</tr>
<tr>
<td></td>
<td>• Entry point offset</td>
</tr>
<tr>
<td></td>
<td>• Statement number: Refers to the line number in the source code (program unit) in which a call was made or an exception took place. The statement number appears only if your routine was compiled with the options required to generate statement numbers. These options are described under “XL C and XL C++ compiler options for AMODE 64 applications” on page 323.</td>
</tr>
<tr>
<td></td>
<td>• Load module: The load module name displayed can be a partitioned data set member or an UNIX executable file. The load module name is also displayed in the third part of the traceback (see below for details).</td>
</tr>
<tr>
<td></td>
<td>• Program unit: The primary entry point of the external procedure. For C routines, this is the compile unit name. For Language Environment-conforming assemblers, this is the ENTNAME = value on the CELQPRLG macro.</td>
</tr>
<tr>
<td></td>
<td>If your routine was compiled with the compile options to generate statement numbers then the program unit name displayed under this column will appear as follows:</td>
</tr>
<tr>
<td></td>
<td>– If your compiled routine is in a partitioned data set then only the member will be output.</td>
</tr>
<tr>
<td></td>
<td>– If your compiled routine is in a sequential data set then only the last qualifier will be shown.</td>
</tr>
<tr>
<td></td>
<td>– If your compiled routine is in an UNIX filename then only what fits of the filename will be displayed in a line.</td>
</tr>
<tr>
<td></td>
<td>Look for the complete name of the program unit in the Fully Qualified Names section of the traceback, if your routine was compiled using compile options to generate statement numbers.</td>
</tr>
<tr>
<td></td>
<td>• Service level: The latest service level applied to the compile unit (for example, for IBM products, it would be the PTF number).</td>
</tr>
<tr>
<td></td>
<td>– If the service level string is equal or less than 7 bytes, all of the string will be output.</td>
</tr>
<tr>
<td></td>
<td>– If the service level string is longer than 7 bytes, the Service column will only show the first 7 bytes of the service string, and the full service string will be shown in section of Full Service Level with max length of 64 bytes.</td>
</tr>
<tr>
<td></td>
<td>• Status: Routine status can be call or exception.</td>
</tr>
</tbody>
</table>
The second part contains:

- DSA number: A number assigned to the information for this active routine by dump processing. The number is used to associate information from the first part of the traceback with information in the second and third parts of the traceback.
- Stack frame (DSA) address
- Entry point address
- Program unit address
- Program unit offset: The offset of the last instruction to run in the routine. If the offset is a negative number, zero, or a very large positive number, the routine associated with the offset probably did not allocate a save area or the routine could have been called using SVC-assisted linkage. Adding the program unit address to the offset gives you the location of the current instruction in the routine. This offset is from the starting address of the routine.
- Compile Date
- Attributes: The attributes of the compile unit including whether character data is being treated as EBCDIC or ASCII and whether floating point data is being treated as IEEE or hexadecimal.

The third part, which is also referred to as 'Fully Qualified Names' section, contains the following:

- DSA number
- Entry
- Program unit: Similar to the Program Unit column in part 1 except that the server name and the complete program unit (PU) name will be displayed. A PU name will appear here only if it was compiled using compile options to produce statement numbers.
- Load Module: The complete pathname of a load module name residing in an UNIX filename will be displayed here if available. The load module's full pathname will be displayed if the PATH environment variable is set such that the pathname of the load module's directory appears before the current directory (.). For load modules found in data sets, the same output shown in the traceback part 1 will also be displayed here.

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

- DSA number
- Entry
- Service: The full service level string with max length of 64 bytes will be displayed here.
<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [6] Condition Information for Active Routines | Displays the following information for all conditions currently active on the call chain:  
- Statement showing failing routine and stack frame address of routine  
- Condition information block (CIB) address  
- The current condition, in the form of a Language Environment message for the condition raised or a Language Environment abend code, if the condition was caused by an abend  
- Location: For the failing routine, this is the program unit, entry routine, statement number, and offset.  
- Machine state, which shows:  
  - Instruction length counter (ILC)  
  - Interruption code  
  - Program status word (PSW)  
  - Contents of GPRs 0–15. Contents of floating point content register (FPC) and floating point registers FPR 0-15.  
  - Storage dump near condition (2 hex-bytes of storage near the PSW)  
  - Storage pointed to by General Purpose Registers  
These values are the current values at the time the condition was raised. |
| [7] Parameters, Registers, and Variables for Active Routines | For each active routine, this section shows:  
- Routine name and stack frame address  
- Saved registers: This lists the contents of GPRs 0–15 at the time the routine received control. The saved registers are those saved by the DSA-owning routine on entry. Register 7 is the return address back to the caller of the DSA-owning routine. Register 6 may be the entry point of the DSA-owning routine. (This is not true when the Branch Relative and Save instruction is used to implement the call. The non-volatile floating-point registers that are saved in the stack frame. The registers are only displayed if the program owning the stack frame saved them. Dashes are displayed in the registers when the register values are not saved.  
- Storage pointed to by the saved registers: Treating the saved contents of each register as an address, 32 bytes before and 64 bytes after the address shown. |
| [8] Control Blocks for Active Routines | For each active routine controlled by the STACKFRAME option, this section lists contents of related control blocks. The Language Environment-conforming language determines which language-specific control blocks appear. The possible control blocks are:  
- Stack frame  
- Condition information block  
- Language-specific control blocks |
| [9] Storage for Active Routines | Displays local storage for each active routine. The storage is dumped in hexadecimal, with EBCDIC translations on the right side of the page. There can be other information, depending on the language used. For C/C++ routines, this is the stack frame storage. |
| [10] Control Blocks Associated with the Thread | Lists the contents of the Language Environment common anchor area (CAA), thread synchronization queue element (SQEL) and dummy stack frame. Other language-specific control blocks can appear in this section. |
Table 53: Contents of the Language Environment dump - AMODE 64. (continued)

<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [11] Enclave Control Blocks | Lists the contents of the Language Environment enclave data block (EDB) and enclave member list (MEML). The information presented may vary depending on which runtime options are set.  
  • If the POSIX runtime option is set to ON, this section lists the contents of the mutex and condition variable control blocks, the enclave level latch table, and the thread synchronization trace block and trace table.  
  • If DLLs have been loaded, this section shows information for each DLL including the DLL name, load address, use count, writable static area (WSA) address, and the thread ID of the thread that loaded the DLL.  
  • If the HEAPCHK runtime option is set to ON, this section shows the contents of the HEAPCHK options control block (HCOP) and the HEAPCHK element tables (HCEL). A HEAPCHK element table contains the location and length of all allocated storage elements for a heap in the order that they were allocated.  
  • When the call-level suboption of the HEAPCHK runtime option is set, any unfreed storage, which would indicate a storage leak, would be displayed in this area. The traceback could then be used to identify the program which did not free the storage.  
  • If the TRACE runtime option is set to ON, this section shows the contents of the Language Environment trace table.  
  Other language-specific control blocks can appear in this section. |
| [12] Runtime Options Report | Lists the Language Environment runtime options in effect when the routine was executed. |
| [13] Process Control Blocks | Lists the contents for the Language Environment process control block (PCB), process member list (MEML), and if the POSIX runtime option is set to ON, the process level latch table. Other language-specific control blocks can appear in this section. |
| [14] CEE3846I CEEDUMP Processing completed. | Identifies the end of the Language Environment dump processing. Similarly, message CEE3845I identifies the start of the dump processing. Message number CEE3846I can be used to locate the end of the previous CEEDUMP report when scanning backward in a data set that contains several CEEDUMP reports. |

Generating a system dump

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

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

**TERMTHDACT(UAONLY, UATRACE, or UADUMP)**
You can use these runtime options, with TRAP(ON), to generate a system dump if an unhandled condition of severity 2 or greater occurs. For 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 337.

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

**Abend Codes in Initialization Assembler User Exit**
Abend codes listed in the initialization assembler user exit are passed to the operating system. The operating system can then generate a system dump.

__cabend()__
You can use the __cabend () API to cause the operating system to handle an abend.
See system or subsystem documentation for detailed system dump information.

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

Steps for generating a system dump in a batch runtime environment

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

1. Specify runtime options TERMTHDACT(UAONLY, UADUMP, UATRACE, or UAIMM), and TRAP(ON). If you specify the suboption UAIMM then you must set TRAP(ON,NOSPIE). The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details on the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 337.

2. Decide whether to include a SYSMDUMP DD card or use the DYNDUMP runtime option.
   - Include a SYSMDUMP DD card with the desired data set name and DCB information:
     LRECL=4160, BLKSIZE=4160, and RECFM=FBS.
   - Specify the DYNDUMP runtime option with the following information:
     `DYNDUMP (hlq,DYNAMIC,TDUMP)`

3. Rerun the program.

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

Perform the following steps to generate a system dump from a z/OS UNIX shell:

• Using _BPXK_MDUMP
   1. Specify where to write the system dump.
      - To write the system dump to a z/OS data set, issue the `export _BPXK_MDUMP=filename` command, where `filename` is a fully qualified data set name with DCB information: LRECL=4160, BLKSIZE=4160, and RECFM=FBS.
        Example: `export _BPXK_MDUMP=hlq.mydump`
      - To write the system dump to an HFS file, issue the `export _BPXK_MDUMP=filename` command, where `filename` is a fully qualified HFS filename:
        Example: `export _BPXK_MDUMP=/tmp/mydump.dmp`
   2. Specify Language Environment runtime options, where `suboption` = UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For more details regarding the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 337.
      `export _CEE_RUNOPTS="termthdact(suboption)"

3. Rerun the program.

When you are done, the system dump is written to the data set name or HFS file name specified. For additional BPXK_MDUMP information, see z/OS UNIX System Services Command Reference.

• Using DYNDUMP
1. Specify Language Environment runtime options:

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

**suboption**

is UAONLY, UADUMP, UATRACE, or UAIMM. If UAIMM is set, TRAP(ON,NOSPIE) must also be set. The TERMTHDACT suboption determines the level of detail of the Language Environment formatted dump. For further details regarding the TERMTHDACT suboptions, see “Generating a Language Environment dump with TERMTHDACT” on page 337

**hlq**

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

2. Rerun the program.

When you are done, the system dump is written to the name generated by the DYNDUMP runtime option. For more DYNDUMP information see [z/OS Language Environment Programming Reference](https://www.ibm.com/support/knowledgecenter/SSEPGG_10.1.0/com.ibm.zos.v1r13.lpar.doc/c_eenv_200.html).

**Note:** You can also specify the signal SIGDUMP on the kill command to generate a system dump of the user address space. For more information about the SIGDUMP signal, see [z/OS UNIX System Services Command Reference](https://www.ibm.com/support/knowledgecenter/SSS77_S/iscf_7.1/insectext/insectext1634.htm).

---

**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](https://www.ibm.com/support/knowledgecenter/SSS77_S/iscf_7.1/insectext/insectext1634.htm).

**Preparing to use the Language Environment support for IPCS**

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

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

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

  ```
  IMBED MEMBER(CEEIPCSP) ENVIRONMENT(IPCS)
  ```

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

- Provide an IPCSPARM DD statement to specify the libraries containing the IPCS control tables; for example:

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

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

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

  ```
  EXIT EP(CEEEANLZ) ANALYZE
  ```

**Understanding Language Environment IPCS VERBEXIT – LEDATA**

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

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

Format

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

Report Type Parameters:

- AUTH
- NTHREADS(value)
- SUM
- HEAP | STACK | SM
- HPT(number) [ HPTTCB (address) ] [ HPTCELL(address) ] [ HPTLOC(location) ]
- CM
- MH
- CEE_DUMP
- COMP(value)
- PTBL(value)
- ALL

Data Selection Parameters:

- DETAIL | EXCEPTION

Control Block Selection Parameters:

- CAA(caa-address)
- DSA(dsa-address)
- TCB(tcb-address)
- ASID(address-space-id)
- NTHREADS(value)
- LAA(laa-address)

Parameters

The following sections describe the various types of parameters you can specify for VERBEXIT LEDATA. Only hexadecimal characters can be specified as addresses provided in LEDATA parameters. Special characters cause the formatter to fail. Therefore, to specify a 64 bit address as a parameter, it must be in the form like 123456789 instead of 1_23456789.

Report type parameters

Use these parameters to select the type of report. If you omit these parameters, the default is SUMMARY.

**Address space report types:** Use these parameters to select a report that shows the Language Environment activity for an address space. Only one of these reports can 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 are displayed. If `value` is specified as asterisk (*), all TCBs are displayed. The LAA, CAA, or TCB parameter can be used to limit the display to only TCBs that are part of the same enclave.

**AUTH**

Requests a report on all Preinitialized Environments for Authorized Programs control blocks for the address space. NTHREADS is ignored when AUTH is specified.
PTBL(value)
Requests that PreInit tables be formatted according to the following values.

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

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

*
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 want.

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

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

HEAP | STACK | SM

HEAP
Requests a report on Storage Management control blocks pertaining to HEAP storage, as well as a detailed report on heap segments. The detailed report includes information about the free storage tree in the heap segment, and information about each allocated storage element. It also specifies a heap pools report with information useful to find potential damaged cells.

Note: Language Environment does not support 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 heap pool trace, if available, be formatted. If the value is 0 or *, the trace for every heap pool ID is formatted. If the value is a single number (1-12), the trace for the specific heap pool ID is formatted. If only the HPT keyword is specified with no value, the trace behaves similar to when the value is *. If no filter is specified, all of the entries are formatted for the specific pool ID.

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

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

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

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

CM
Requests a report on Condition Management control blocks.

MH
Requests a report on Message Handler control blocks.

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

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

C  Requests a report on C/C++ runtime control blocks.
CIO  Requests a report on C/C++ I/O control blocks.
COBOL  Requests a report on COBOL-specific control blocks.
PLI  Requests a report on PL/I-specific control blocks.
ALL  Requests a report on all the previous control blocks.

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

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

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

Data selection parameters
Data selection parameters limit the scope of the data in the report. If no data selection parameter is selected, the default is DETAIL.

DETail
Requests formatting all control blocks for the selected components. Only significant fields in each control block are formatted. For the Heap and Storage Management Reports, the DETAIL parameter will provide a detailed heap segment report for each heap segment in the dump. The detailed heap
segment report includes information on the free storage tree in the heap segments, and all allocated storage elements. This report will also identify problems detected in the heap management data structures. For more information about the Heap Reports, see “Understanding the HEAP LEDATA output” on page 375.

EXCeption
Requests validating all control blocks for the selected components. Output is only produced naming the control block and its address for the first control block in a chain that is invalid. Validation consists of control block header verification at the very least. For the Summary, CEEDUMP, C/C++, PL/I reports, the EXCEPTION parameter has not been implemented. For these reports, DETAIL output is always produced.

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

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

DSA(dsa-address)
specifies the address of the DSA. If not specified, the DSA address may be obtained from the TCB or the IPCS symbol REGGEN.

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

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

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

Examples
For examples of the output produced by LEDATA and explanation of the content, refer to “Understanding the Language Environment IPCS VERBEXIT LEDATA output” on page 360.

Understanding the Language Environment IPCS VERBEXIT LEDATA output
The Language Environment IPCS VERBEXIT LEDATA generates formatted output of the Language Environment runtime environment control blocks from a system dump. The following sample illustrates the output produced when the LEDATA VERBEXIT is invoked with the ALL parameter. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option when running the program CELQSAMP in Figure 162 on page 340.

“Sections of the Language Environment LEDATA VERBEXIT formatted output” on page 370 describes the information in the formatted output. Ellipses are used to summarize some sections of the dump. For easy
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Registers and PSW:**

- **GPR0**
- **GPR1**
- **GPR2**
- **GPR3**
- **GPR4**
- **GPR5**
- **GPR6**
- **GPR7**
- **GPR8**
- **GPR9**
- **GPR10**
- **GPR11**
- **GPR12**
- **GPR13**
- **GPR14**
- **GPR15**

---

**PCB Address:** 00000001_00003CA0

**CAA Address:** 00000001_00007B18

**TCB Address:** 007FF050

---

**Language Environment Product 84 V01 R99.00**

---

**Traceback:**

1. **CEEDMPP** +0000099A
2. **CEEDISP** +00000384
3. **CEEDISP** +00000384
4. **CEEDISP** +00000384
5. **CEEDISP** +00000384
6. **CEEDISP** +00000384
7. **div_zero** +0000004A
8. **main** +0000004A
9. **CELOMERT** +0000134A

---

**Mutex and Condition Variable Blocks (MCVB+MHT+CHT):** 00000001_089100B8

---

**Control Blocks Associated with the Thread:**

- **Thread Synchronization Queue Element (SOEL):** 00000000_257520B0

---

**Enclave Control Blocks:**

- **Mutex and Condition Variable Blocks (MCVB+MHT+CHT):** 00000000_089100B8

---

**Using Language Environment AMODE 64 debugging facilities**

---

*Figure 174: Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 10)*
The following is part two of the example of formatted output from LEDATA VERBEXIT (AMODE 64).

<table>
<thead>
<tr>
<th>Displacement Trace Entry in Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Synchronization Enclave Latch Table (EPALT): 00000001_08910BB0</td>
</tr>
<tr>
<td>+00000 00000001_08910BB0 00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+00005 00000001_08910BB5 00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0000A 00000001_08910BB1 00000000 00000000 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

The following is part two of the example of formatted output from LEDATA VERBEXIT (AMODE 64).
The following is part three of the example of formatted output from LEDATA VERBEXIT (AMODE 64).

Figure 176: Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 3 of 10)
The following is part four of the example of formatted output from LEDATA VERBEXIT (AMODE 64).

**Figure 177: Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 4 of 10)**
The following is part five of the example of formatted output from LEDATA VERBEXIT (AMODE 64).

Using Language Environment AMODE 64 debugging facilities 365
Heap Storage Control Blocks

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

ENSQ: 00000001_00100108

+000000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

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

00000001_001008 ENSQ: 00000001_00100108

+000000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENSQ HEAPPOLS trace available. To display: IP VERBX LEDATA 'HPT(*)'

ENSQ: 00000001_00100108

+080000 Heap-control:ENQ...
Figure 180: Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 7 of 10)
The following is part eight of the example of formatted output from LEDATA VERBEXIT (AMODE 64).

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSA: 00000001_082FAAC0</td>
<td></td>
</tr>
<tr>
<td>+000800  HPR4:00000001 082FD3E0</td>
<td>HPR5:00000000 251BABA0</td>
</tr>
<tr>
<td>+000810  HPR6:00000000 251B6060</td>
<td>HPR7:00000000 2504B400</td>
</tr>
<tr>
<td>+000820  HPR8:00000001 089135B0</td>
<td>HPR9:00000000 00000005</td>
</tr>
<tr>
<td>+000830  HPR10:00000001 082FE3DF</td>
<td>HPR11:00000001 082FE0C0</td>
</tr>
<tr>
<td>+000840  HPR12:00000001 089135B0</td>
<td>HPR13:00000001 082FE680</td>
</tr>
<tr>
<td>+000850  HPR14:00000000 2504B300</td>
<td>HPR15:00000000 2530A300</td>
</tr>
<tr>
<td>+000860  HPHKSAV:00000000 00000000</td>
<td>HPTRAN:00000000 00000000</td>
</tr>
<tr>
<td>+000878  HPRENT:00000000 00000000</td>
<td></td>
</tr>
</tbody>
</table>

Contents of DSA at Location : 00000001_082FB2C0

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 00000001_082FB2C0</td>
<td>00000001 082FD3E0 00000000 251BABA0</td>
</tr>
<tr>
<td>+000010 00000001_082FB2D0</td>
<td>00000000 251B6060 00000000 2504B400</td>
</tr>
<tr>
<td>+000020 00000001_082FB2E0</td>
<td>00000001 089135B0 00000000 00000005</td>
</tr>
<tr>
<td>+000030 00000001_082FB2F0</td>
<td>00000001 082FE3DF 00000001 082FE0C0</td>
</tr>
<tr>
<td>+000040 00000001_082FB300</td>
<td>00000001 089135B0 00000001 082FE680</td>
</tr>
<tr>
<td>+000050 00000001_082FB310</td>
<td>00000000 2504B300 00000000 2530A300</td>
</tr>
<tr>
<td>+000060 00000001_082FB320</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+000070 00000001_082FB330</td>
<td>+000000 00000001_082FB33F</td>
</tr>
<tr>
<td>+000080 00000001_082FB340</td>
<td>00000001 082FBB08 00000001 00000003</td>
</tr>
<tr>
<td>+000090 00000001_082FB350</td>
<td>00000001 00000003 00000001 00007208</td>
</tr>
<tr>
<td>+0000A0 00000001_082FB360</td>
<td>00000001 082FC190 00000001 082FBF90</td>
</tr>
<tr>
<td>+0000B0 00000001_082FB370</td>
<td>00000000 25773048 00000000 00000000</td>
</tr>
<tr>
<td>+0000C0 00000001_082FB380</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+0000D0 00000001_082FB390</td>
<td>+000000</td>
</tr>
</tbody>
</table>

Contents of DSA at Location : 00000001_082FDBE0

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 00000001_082FDBE0</td>
<td>00000001 082FDDE0 00000000 2504C3EC</td>
</tr>
<tr>
<td>+000010 00000001_082FDBF0</td>
<td>00000000 2504AAB0 00000000 251C96D0</td>
</tr>
<tr>
<td>+000020 00000001_082FDC00</td>
<td>00000000 25754AD8 00000000 25754BD0</td>
</tr>
<tr>
<td>+000030 00000001_082FDC10</td>
<td>00000000 25754A30 00000000 00000020</td>
</tr>
<tr>
<td>+000040 00000001_082FDC20</td>
<td>00000001 00007B18 00000001 082FE680</td>
</tr>
<tr>
<td>+000050 00000001_082FDC30</td>
<td>00000000 253D6504 00000000 00000003</td>
</tr>
</tbody>
</table>

Condition Management Control Blocks

HCOM: 00000000_25753700

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000 PICA_AREA:00000000 00000000</td>
<td>EYES:HCOM SIZE:28E0 LEVEL:0001</td>
</tr>
<tr>
<td>+000010 CAA_PTR1:00000001_00007B18</td>
<td>CVTDCB:9B FLAG1:60104000</td>
</tr>
<tr>
<td>+000020 EXIT_STK:00000000_00000000</td>
<td>RSM_PTR:00000000_00000000</td>
</tr>
<tr>
<td>+000030 HDLL_STK:00000000_00000000</td>
<td>SRP_TOKEN:00000000_00000000</td>
</tr>
<tr>
<td>+000040 CURR_STK:00000000_00000000</td>
<td>CIBH:00000001_082FCFE0</td>
</tr>
<tr>
<td>+000050 SPIE_TOKEN:00000000</td>
<td>DMCP:00000000_00000000</td>
</tr>
<tr>
<td>+000100 COND_LOG:00000001_08FE9910</td>
<td>4083_DSA:00000000_00000000</td>
</tr>
<tr>
<td>+000704 SHUNT_COUNTER:00000000</td>
<td>SHUNT_VALIDFLAG:00000000</td>
</tr>
<tr>
<td>+000710 SHUNT_PSW:00000000 00000000 00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000720 SHUNT_REG0:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000728 SHUNT_REG1:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000730 SHUNT_REG2:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000738 SHUNT_REG3:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000740 SHUNT_REG4:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000748 SHUNT_REG5:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000750 SHUNT_REG6:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000758 SHUNT_REG7:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000760 SHUNT_REG8:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000768 SHUNT_REG9:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000770 SHUNT_REG10:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000778 SHUNT_REG11:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000780 SHUNT_REG12:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000788 SHUNT_REG13:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000790 SHUNT_REG14:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+000798 SHUNT_REG15:00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>+0007A8 SHUNT_CODE1:00000000_00000000</td>
<td></td>
</tr>
</tbody>
</table>

Figure 181: Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 8 of 10)
The following is part nine of the example of formatted output from LEDATA VERBEXIT (AMODE 64).

Figure 182: Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 9 of 10)
The following is part ten of the example of formatted output from LEDATA VERBEXIT (AMODE 64).

Figure 183: Example of formatted output from LEDATA VERBEXIT (AMODE 64) (Part 10 of 10)

Sections of the Language Environment LEDATA VERBEXIT formatted output

Table 54 on page 371 lists the sections of the LEDATA VERBEXIT output, which appear independently of the Language Environment-conforming languages used.

370 z/OS: Language Environment Debugging Guide
Table 54: Contents of the Language Environment LEDATA VERBEXIT formatted output (AMODE 64).

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] - [8] CEEDUMP Formatted Control Blocks: These sections are included when the CEEDUMP parameter is specified on the LEDATA invocation.</td>
<td></td>
</tr>
<tr>
<td>[1] - [4] 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.</td>
<td></td>
</tr>
<tr>
<td>[1] Enclave Identifier</td>
<td>Names the enclave for which information is provided.</td>
</tr>
<tr>
<td>[2] Information for thread</td>
<td>Shows the system identifier for the thread. Each thread has a unique identifier.</td>
</tr>
<tr>
<td>[3] Registers and PSW</td>
<td>Displays the register and program status word (PSW) values that were used to create the traceback. These values may come from the TCB, the RTM2 work area, a linkage stack entry or output from the BPXGMSTA service. This section is not displayed when the DSA() parameter is specified on the LEDATA invocation.</td>
</tr>
<tr>
<td>[4] Traceback</td>
<td>For all active routines in a particular thread, the traceback section shows routine information in two parts. The first part contains:</td>
</tr>
</tbody>
</table>

- 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 PL/I routines, this is the entry point name. For C/C++ routines, this is the function name. If a function name or entry point was not specified for a particular routine, then the string ** NoName ** will appear. |

  - Entry point offset |
  - Load module |

- Program unit: The primary entry point of the external procedure. For C and PL/I routines, this is the compile unit name. For Language Environment-conforming assemblers, this is the ENTNAME = value on the CELQPRLG macro. |

  - Service level: The latest service level applied to the compile unit (for example, for IBM products, it would be the PTF number. |

    - If the service level string is equal or less than 7 bytes, all of the string will be output. |

    - If the service level string is longer than 7 bytes, the Service column will only show the first 7 bytes of the service string, and the full service string will be shown in section of Full Service Level with max length of 64 bytes. |

- Status: Routine status can be call, exception, or running. |

The second part contains: |

- 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. |

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

- DSA number |

- Entry |

- Service: The full service level string with max length of 64 bytes will be displayed here.
Table 54: Contents of the Language Environment LEDATA VERBEXIT formatted output (AMODE 64). (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Control Blocks Associated with the Thread</td>
<td>Lists the contents of the thread synchronization queue element (SQEL).</td>
</tr>
<tr>
<td>6 Enclave Control Blocks</td>
<td>If the POSIX runtime option was set to ON, this section lists the contents of the mutex and condition variable control blocks, the enclave level latch table, and the thread synchronization trace block and trace table. If the HEAPCHK runtime option is set to ON, this section lists the contents of the HEAPCHK options control block (HCOP) and the HEAPCHK element tables (HCEL). A HEAPCHK element table contains the location and length of all allocated storage elements for a heap in the order that they were allocated.</td>
</tr>
<tr>
<td>7 Language Environment Trace Table</td>
<td>If the TRACE runtime option was set to ON, this section shows the contents of the Language Environment trace table.</td>
</tr>
<tr>
<td>8 Process Control Blocks</td>
<td>If the POSIX runtime option was set to ON, this section lists the contents of the process level latch table.</td>
</tr>
<tr>
<td>9 - 17 Summary: These sections are included when the SUMMARY parameter is specified on the LEDATA invocation.</td>
<td></td>
</tr>
<tr>
<td>9 Summary Header</td>
<td>The summary header section contains:</td>
</tr>
<tr>
<td></td>
<td>• Address of Thread control block (TCB)</td>
</tr>
<tr>
<td></td>
<td>• Release number</td>
</tr>
<tr>
<td></td>
<td>• Address Space ID (ASID)</td>
</tr>
<tr>
<td>10 Active Members List</td>
<td>Lists active members, which is extracted from the enclave member list (MEML).</td>
</tr>
<tr>
<td>11 CEELAA</td>
<td>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.</td>
</tr>
<tr>
<td>12 CEELCA</td>
<td>Formats the contents of the Language Environment library control area (LCA). See z/OS Language Environment Vendor Interfaces for a description of the fields in the LCA.</td>
</tr>
<tr>
<td>13 CEECAA</td>
<td>Formats the contents of the Language Environment common anchor area (CAA). See z/OS Language Environment Vendor Interfaces for a description of the fields in the CAA. If there is any, DLL failure data is also formatted.</td>
</tr>
<tr>
<td>14 CEEPCB</td>
<td>Formats the contents of the Language Environment process control block (PCB), and the process level member list.</td>
</tr>
<tr>
<td>15 CEERCB</td>
<td>Formats the contents of the Language Environment region control block (RCB).</td>
</tr>
<tr>
<td>16 CEEEDB</td>
<td>Formats the contents of the Language Environment enclave data block (EDB), and the enclave level member list.</td>
</tr>
<tr>
<td>17 Runtime Options</td>
<td>Lists the runtime options in effect at the time of the dump, and indicates where they were set.</td>
</tr>
<tr>
<td>18 Heap Storage Control Blocks</td>
<td>This section is included when the HEAP or SM parameter is specified on the LEDATA invocation. It formats the Enclave-level storage management control block (ENSQ) and for each different type of heap storage:</td>
</tr>
<tr>
<td></td>
<td>• Heap control block (HPCQ)</td>
</tr>
<tr>
<td></td>
<td>• Chain of heap anchor blocks (HANQ). A HANQ immediately precedes each segment of heap storage.</td>
</tr>
<tr>
<td></td>
<td>This section includes a detailed heap segment report for each segment in the dump. For more information about the detailed heap segment report, see “Understanding the HEAP LEDATA output” on page 375.</td>
</tr>
</tbody>
</table>
Table 54: Contents of the Language Environment LEDATA VERBEXIT formatted output (AMODE 64). (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [19] Stack Storage Control Blocks | This section is included when the STACK or SM parameter is specified on the LEDATA invocation. This section formats:  
  - Stack anchor (SANC)  
  - Chain of dynamic save areas (DSA) |
| [20] Condition Management Control Blocks | This section is included when the CM parameter is specified on the LEDATA invocation. It formats the chain of Condition Information Block Headers (CIBH) and Condition Information Blocks. The Machine State Information Block is contained with the CIBH starting with the field labeled MCH_EYE. |
| [21] Message Processing Control Blocks | This section is included when the MH parameter is specified on the LEDATA invocation. |
| [22] Preinitialization Information | This section is included when the PTBL parameter is specified on the LEDATA invocation. It formats information related to preinitialization. See “PTBL LEDATA output” on page 373 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 184 on page 374 illustrates the output produced when the VERBEXIT LEDATA command is invoked with the PTBL parameter.
PTBL(CURRENT)

Language Environment Product 04 V01 R09.00

PreInitialization Programming Interface Trace Data

CELQPIPI Environment Table Entry and Trace Entry:
Active CELQPIPI Environment (Address 00000001_00010B80)
Eyecatcher: CELQIPTB
TCB address: 008D6E88

CELQPIPI Environment:
Environment Type: MAIN
Sequence of Calls not active
Exits not established
Signal Interrupt Routines not registered
Service Routines are not active

CELQPIPI Environment Enclave Initialized
Number of CELQPIPI Table Entries = 3

CELQPIPI Table Entry Information:

<table>
<thead>
<tr>
<th>CELQPIPI Table Index</th>
<th>Routine Name</th>
<th>Routine Type</th>
<th>Routine Entry Point</th>
<th>Routine Function Pointer</th>
<th>Routine was loaded by Language Environment</th>
<th>Routine Address was resolved</th>
<th>Routine Function Descriptor was valid</th>
<th>Routine was valid</th>
<th>Routine Return Code</th>
<th>Routine Reason Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ISJPPCA3</td>
<td>C/C++</td>
<td>00000000_21053000</td>
<td>00000000_210530C0</td>
<td>Routine was loaded by Language Environment</td>
<td>Routine Address was resolved</td>
<td>Routine Function Descriptor was valid</td>
<td>Routine was valid</td>
<td>Routine Return Code</td>
<td>Routine Reason Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Entry of routine in CELQPIPI Table for Index 0 (00000001_00010D38)

CELQPIPI Trace Table Entries:

<table>
<thead>
<tr>
<th>Call Type</th>
<th>Routine Table Index</th>
<th>Routine Programming Language</th>
<th>Service RC</th>
<th>Routine was added to the PreInit table.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT_MAIN</td>
<td>1</td>
<td>C/C++</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ADD_ENTRY</td>
<td>1</td>
<td>C/C++</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>IDENTIFY_ENTRY</td>
<td>1</td>
<td>C/C++</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CALL_MAIN</td>
<td>1</td>
<td>C/C++</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>DELETE_ENTRY</td>
<td>1</td>
<td>C/C++</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Exiting Language Environment Data

Figure 184: Example of formatted PTBL output from LEDATA VERBEXIT (Part 1 of 2)

Figure 185: Example of formatted PTBL output from LEDATA VERBEXIT (Part 2 of 2)
Understanding the HEAP LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap segment report when the HEAP option is used with the DETAIL option, or when the SM,DETAIL option is specified. The detailed heap segment report is useful when trying to pinpoint damage because it provides specific information. The report describes the nature of the damage, and specifies where the actual damage occurred. The report can also be used to diagnose storage leaks, and to identify heap fragmentation. Figure 186 on page 375 shows the output produced by specifying the HEAP option. “Heap report sections of the LEDATA output” on page 382 describes the information in the formatted output.

For easy reference, the sections of the dump are numbered to correspond with the description of each section that follows. Ellipses are used to summarize some sections of the dump.

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

---

**Figure 186: Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 1 of 8)**
Map of Heap Segment 0000000108300000
To display entire segment: IP LIST 0000000108300000 LEN(X'0000000000100000') ASID(X'0028')
0000000108300040: Allocated storage element, length=0000000000000180.
To display: IP LIST 0000000108300040 LEN(X'0000000000000180') ASID(X'0028')
0000000108300050: C36DE6E2 C1F6F440 40404040 40404040  00000001 08300070 00000000 250005A8 |C_WSA64         ...............y|
00000001083001C0: Allocated storage element, length=0000000000019660.
To display: IP LIST 00000001083001C0 LEN(X'0000000000019660') ASID(X'0028')
00000001083001D0: 00000000 00000005 00000000 00000005  00000001 08319840 00000001 08319A28 |......................q ........|
⋮
00000001083B79A0: Allocated storage element, length=0000000000017720.
To display: IP LIST 00000001083B79A0 LEN(X'0000000000017720') ASID(X'0028')
00000001083B79B0: 00000000 00000002 00000000 00000000  00000000 00000000 00000000 D3D4D9C5 |............................LMRE|
00000001083CF0C0: Free storage element, length=0000000000030F40.
To display: IP LIST 00000001083CF0C0 LEN(X'0000000000030F40') ASID(X'0028')
Summary of analysis for Heap Segment 0000000108300000:
Amounts of identified storage:  Free:00030F40          Allocated:000CF080          Total:000FFFC0
Number of identified areas   :  Free:       1  Allocated:      12  Total:      13
00000000 bytes of storage were not accounted for.
No errors were found while processing this heap segment.
THNQ:  00000001_001007F0
+000000  EYE_CATCHER:THNQ  FLAGS:00000000    NEXT:00000001_00100138
+000010  PREV:00000001_001005B0        HEAPID:00000001_00100138
+000020  SEGMENT:00000001_19C00000     SEG_LEN:00000000 00100000
HANQ:  00000001_19C00000
+000000  EYE_CATCHER:HANQ  FLAGS:00000000    HEAPID:00000001_00100138
+000020  SEGMENT:00000001_19C00000     ROOT:00000001_19C00040
+000030  SEG_LEN:00000000 00100000     ROOT_LEN:00000000 000FFFC0
This is the last heap segment in the current heap.

Free Storage Tree for Heap Segment 0000000119C00000
Node              Node             Parent            Left            Right                Left              Right
Depth      Address            Length             Node             Node            Node                Length            Length
0  0000000119C00040  00000000000FFFC0  0000000000000000  0000000000000000  0000000000000000  0000000000000000

Free Storage Tree for Heap Segment 0000000119C00000
To display entire segment: IP LIST 0000000119C00000 LEN(X'0000000000100000') ASID(X'0028')
0000000119C00040: Free storage element, length=00000000000FFFC0.
To display: IP LIST 0000000119C00040 LEN(X'00000000000FFFC0') ASID(X'0028')
Summary of analysis for Heap Segment 0000000119C00000:
Amounts of identified storage:  Free:000FFFC0          Allocated:00000000          Total:000FFFC0
Number of identified areas   :  Free:       1  Allocated:       0  Total:       1
00000000 bytes of storage were not accounted for.
No errors were found while processing this heap segment.
This is the last heap segment in the current heap.
Library Heap64 Control Blocks
HPCQ:  00000001_00100168
+000000  EYE_CATCHER:HPCQ  FIRST:00000001_001005E0       LAST:00000001_00100670
+000018  INITSIZE:00000000 00000001    INCRSIZE:00000000 00000001
+00002C  OPTIONS:90000000
HPSQ:  00000001_000050E8
+000000  BYTES_ALLOC:00000000 0067A940
+000008  CURR_ALLOC:00000000 0067A840        GET_REQ:00000000 00000042
+000018  FREE_REQ:00000000 00000003    GETMAINS:00000000 00000003
+000028  FREEMAINS:00000000 00000000
THNQ:  00000001_001005E0
+000000  EYE_CATCHER:THNQ  FLAGS:80000000    NEXT:00000001_00100610
+000010  PREV:00000001_00100168        HEAPID:00000001_00100168
+000020  SEGMENT:00000001_08400000     SEG_LEN:00000000 00100000
HANQ:  00000001_08400000
+000000  EYE_CATCHER:HANQ  FLAGS:80000000    HEAPID:00000001_00100168
+000020  SEGMENT:00000001_08400000     ROOT:00000001_08400040
+000030  SEG_LEN:00000000 00100000     ROOT_LEN:00000000 000FFFC0

Figure 187: Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 2 of 8)
The following is part three of the example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64).

Figure 188: Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 3 of 8)
The following is part four of the example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64).

**Figure 189: Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 4 of 8)**
The following is part five of the example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64).

<table>
<thead>
<tr>
<th>Library Heap31 Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPCQ: 00000001_001001IC</td>
</tr>
<tr>
<td>+000000 BYTEALOC:00000000 00000000</td>
</tr>
<tr>
<td>+000000 CORRALLOC:00000000 00000000 GET_REG:00000080 00000080</td>
</tr>
<tr>
<td>+000018 FREE_REG:00000000 00000000 GETMAINS:00000000 00000000</td>
</tr>
<tr>
<td>+000028 FREEMAINS:00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library Heap31 Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPSQ: 00000001_00005088</td>
</tr>
<tr>
<td>+000000 BYTES_ALLOC:00000000 00000000</td>
</tr>
<tr>
<td>+000008 CURR_ALLOC:00000000 00000000</td>
</tr>
<tr>
<td>+000016 GET_REQ:00000000 00000000</td>
</tr>
<tr>
<td>+000024 FREE_REQ:00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library Heap31 Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>THNQ: 00000001_001007C0</td>
</tr>
<tr>
<td>+000000 EYE_CATCHER:THNQ</td>
</tr>
<tr>
<td>+000018 INITSIZE:00000000 00004000</td>
</tr>
<tr>
<td>+00002C OPTIONS:50000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library Heap31 Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>HANQ: 00000000_25773000</td>
</tr>
<tr>
<td>+000000 EYE_CATCHER:HANQ</td>
</tr>
<tr>
<td>+000020 SEGMENT:25773000</td>
</tr>
<tr>
<td>+00002C ROOT:25774168</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library Heap24 Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPCQ: 00000001_001001F8</td>
</tr>
<tr>
<td>+000000 BYTEALOC:00000000 00000000</td>
</tr>
<tr>
<td>+000000 CORRALLOC:00000000 00000000</td>
</tr>
<tr>
<td>+000000 GET_REG:00000080 00000080</td>
</tr>
<tr>
<td>+000018 FREE_REG:00000000 00000000</td>
</tr>
<tr>
<td>+000024 GETMAINS:00000000 00000000</td>
</tr>
<tr>
<td>+00002C OPTIONS:30000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library Heap24 Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPSQ: 00000001_000050B8</td>
</tr>
<tr>
<td>+000000 BYTES_ALLOC:00000000 00000000</td>
</tr>
<tr>
<td>+000008 CURR_ALLOC:00000000 00000000</td>
</tr>
<tr>
<td>+000016 GET_REQ:00000000 00000000</td>
</tr>
<tr>
<td>+000024 FREE_REQ:00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library Heap24 Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>THNQ: 00000001_001007C0</td>
</tr>
<tr>
<td>+000000 EYE_CATCHER:THNQ</td>
</tr>
<tr>
<td>+000018 OPTIONS:30000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library Heap24 Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>HANQ: 00000000_25773000</td>
</tr>
<tr>
<td>+000000 EYE_CATCHER:HANQ</td>
</tr>
<tr>
<td>+000020 SEGMENT:25773000</td>
</tr>
<tr>
<td>+00002C ROOT:25774168</td>
</tr>
</tbody>
</table>

** NO SEGMENTS ALLOCATED **

Free Storage Tree for Heap Segment 25773000

<table>
<thead>
<tr>
<th>Node</th>
<th>Node</th>
<th>Parent</th>
<th>Left</th>
<th>Right</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25774168</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
</tbody>
</table>

Map of Heap Segment 25773000

To display entire segment: IP LIST 25773000 LEN(X'00004000') ASID(X'0028')

25773040: Allocated storage element, length=00001128. To display: IP LIST 25773040 LEN(X'00001128') ASID(X'0028')

25774168: Free storage element, length=00002E98. To display: IP LIST 25774168 LEN(X'00002E98') ASID(X'0028')

Summary of analysis for Heap Segment 25773000:

Amounts of identified storage: Free:00000000 Allocated:00001128 Total:00001128
Number of identified areas Free: 1 Allocated: 1 Total: 2
000000 bytes of storage were not accounted for.
No errors were found while processing this heap segment.
This is the last heap segment in the current heap.

User Heap24 Control Blocks

| HPCQ: 00000001_001001F8 |
| +000000 BYTEALOC:00000000 00000000 |
| +000000 CORRALLOC:00000000 00000000 |
| +000000 GET_REG:00000080 00000080 |
| +000018 FREE_REG:00000000 00000000 |
| +000024 GETMAINS:00000000 00000000 |
| +00002C OPTIONS:30000000 |

** NO SEGMENTS ALLOCATED **

Library Heap24 Control Blocks

| HPCQ: 00000001_00100828 |
| +000000 BYTEALOC:00000000 00000000 |
| +000000 CORRALLOC:00000000 00000000 |
| +000000 GET_REG:00000080 00000080 |
| +000018 FREE_REG:00000000 00000000 |
| +000024 GETMAINS:00000000 00000000 |
| +00002C OPTIONS:30000000 |

Figure 190: Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 5 of 8)
The following is part six of the example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64).

Figure 191: Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 6 of 8)
The following is part seven of the example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64).

Figure 192: Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 7 of 8)
The following is part eight of the example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64).

Figure 193: Example formatted detailed heap segment report from LEDATA VERBEXIT (AMODE 64) (Part 8 of 8)

Heap report sections of the LEDATA output

The Heap Report sections of the LEDATA output provide information for each heap segment in the dump. The detailed heap segment reports include information on the free storage tree in the heap segments, the allocated storage elements, and the cause of heap management data structure problems.
<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
</table>
| [1] Free Storage Tree Report | Within each heap segment, Language Environment keeps track of unallocated storage areas by chaining them together into a tree. Each free area represents a node in the tree. Each node contains a header, which points to its left and right child nodes. The header also contains the length of each child. The LEDATA HEAP option formats the free storage tree within each heap, and validates all node addresses and lengths within each node. Each node address is validated to ensure that it:  
  • Falls on a doubleword boundary  
  • Falls within the current heap segment  
  • Does not point to itself  
  • Does not point to a node that was previously traversed  
Each node length is validated to ensure that it:  
  • Is a multiple of 8  
  • Is not larger than the heap segment length  
  • Does not cause the end of the node to fall outside of the current heap segment  
  • Does not cause the node to overlap another node  
If the formatter finds a problem, then it will place an error message describing the problem directly after the formatted line of the node that failed validation. |
| [2] Heap Segment Map Report | The LEDATA HEAP option produces a report that lists all of the storage areas within each heap segment, and identifies the area as either allocated or freed. For each allocated area the contents of the first X’20’ bytes of the area are displayed in order to help identify the reason for the storage allocation. Each allocated storage element has a prefix used by Language Environment to manage the area. The prefix contains a pointer to the start of the heap segment followed by the length of the allocated storage element. For HEAP64 heaps, the prefix is 16 bytes, with 8-byte pointer and length fields. For HEAP31 and HEAP24 heaps, the pointer is 8 bytes with 4-byte pointer and length field. The formatter validates this header to ensure that its heap segment pointer is valid. The length is also validated to ensure that it:  
  • Is a multiple of 8  
  • Is not zero  
  • Is not larger than the heap segment length  
  • Does not cause the end of the element to fall outside of the current heap segment  
  • Does not cause the element to overlap a free storage node  
If the heap_free_value of the STORAGE runtime option was specified, then the formatter also checks that the free storage within each free storage element is set to the requested heap_free_value. If a problem is found, then an error message describing the problem is placed after the formatted line of the storage element that failed validation. |

**Diagnosing heap damage problems**

Heap storage errors can occur when an application allocates a heap storage element that is too small for it to use, and therefore, accidently overlays heap storage. If this situation occurs then some of the typical error messages generated are:

- The node address does not represent a valid node within the heap segment
- The length of the segment is not valid, or
- The heap segment pointer is not valid.

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

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

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

**Diagnosing storage leak problems**

A storage leak occurs when a program does not return storage back to the heap after it has finished using it. To determine if this problem exists, do one of the following:

- The **call-level** suboption of the HEAPCHK runtime option causes a report to be produced in the CEEDUMP. Any still-allocated (that is, not freed) storage identified by HEAPCHK is listed in the report, along with the corresponding traceback. This shows any storage that wasn't freed, as well as all the calls that were involved in allocating the storage. For more information about the HEAPCHK runtime option, see z/OS Language Environment Programming Reference.

- Examine the Heap Segment Map report to see if any data areas, within the allocated storage elements, appear more frequently than expected. If they do, then check to see if these data areas are still being used by the application program. If the data areas are not being used, then change the program to free the storage element after it is done with it.

**Diagnosing heap fragmentation problems**

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

**Understanding the heap pool LEDATA output**

The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap pool report when HEAPPOOLS is ON. The detailed heap pool report is useful when trying to find potential damaged cells because it provides very specific information. Figure 194 on page 385 illustrates the details of heap pool report. “Heap pool report sections of the LEDATA output” on page 389 describes the information contained in the formatted output.
Heap Pool Report
QPCB: 00000000_08733600
+000000 EYECATCHER:QPCB LENGTH:00001800 NUMPOOLS:00000010
+000000C LARGEST_CELL_SIZE:0000001000 BIG_REQUESTS:00000000
+000000B STORAGE_HITS_ADDR:00000000 00000005 Flags:0400
+000022 NUMTARRAYS:05 NUMCELLSIZES:0C
+00002B GET_POOLINFO_ARRAYS_PTR:00000000_08733900

Data for pool 1:
POOLDATA: 00000000_08733000
+000000 POOL_INDEX:00000001 INPUT_CELL_SIZE:00000008
+000008 CELL_SIZE:0000001000 INPUT_COUNT:0000000A
+000010 CELL_POOL_SIZE:00000140 CELL_POOL_NUM:0000000A
+000018 LAST_CELL:00000000_0862E6B0 NEXT_CELL:00000000_0862E6B0
+000040 Q_CONTROL_INFO:00000000 00000005 Q_FIRST_CELL:00000000_0862E680
+000050 POOL_NUM_GET_TOTAL:00000000 00000003
+000058 POOL_NUM_FREE:00000000 00000001 POOL_EXTENTS_ANCHOR:00000000_0862E6B0
+000068 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:02
+00006A POOL_NUM_SAME_SIZE:01 POOL_TRACE_TABLE:00000000_0886C1A0

EXTENT: 00000000_0862E6B0
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_00000000
To display entire pool extent: IP LIST 00000000_0862E6B0 LEN(X'000000D0') ASID(X'0021')
000000080862E6C0: Free storage cell. To display: IP LIST 000000080862E6C0 LEN(X'00000030') ASID(X'0021')

[1] Verifying free chain for pool: 1...
No errors were found while processing free chain.
Summary of analysis for Pool 1:
Number of cells: Unused: 9 Free: 1 Allocated: 0 Total Used: 10
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 2:
POOLDATA: 00000000_08733000
+000000 POOL_INDEX:00000002 INPUT_CELL_SIZE:00000020
+000008 CELL_SIZE:00000030 INPUT_COUNT:00000004
+000010 CELL_POOL_SIZE:000000C0 CELL_POOL_NUM:00000004
+000018 LAST_CELL:00000000_0862E750 NEXT_CELL:00000000_0862E750
+000040 Q_CONTROL_INFO:00000000 00000005 Q_FIRST_CELL:00000000_0862E750
+000050 POOL_NUM_GET_TOTAL:00000000 0000000E
+000058 POOL_NUM_FREE:00000000 00000002 POOL_EXTENTS_ANCHOR:00000000_0862E750
+000068 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:03
+00006A POOL_NUM_SAME_SIZE:01 POOL_TRACE_TABLE:00000000_0886C1A0

EXTENT: 00000000_0862E750
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_00000000
To display entire pool extent: IP LIST 00000000_0862E750 LEN(X'000000D0') ASID(X'0021')
000000080862E760: Free storage cell. To display: IP LIST 000000080862E760 LEN(X'00000030') ASID(X'0021')

[1] Verifying free chain for pool: 2...
No errors were found while processing free chain.
Summary of analysis for Pool 2:
Number of cells: Unused: 3 Free: 1 Allocated: 0 Total Used: 4
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

Data for pool 3:
POOLDATA: 00000000_08733000
+000000 POOL_INDEX:00000003 INPUT_CELL_SIZE:00000080
+000008 CELL_SIZE:00000090 INPUT_COUNT:00000004
+000010 CELL_POOL_SIZE:00000240 CELL_POOL_NUM:00000004
+000018 LAST_CELL:00000000_0862E950 NEXT_CELL:00000000_0862E950
+000040 Q_CONTROL_INFO:00000000 00000010 Q_FIRST_CELL:00000000_0862E830
+000050 POOL_NUM_GET_TOTAL:00000000 0000000E
+000058 POOL_NUM_FREE:00000000 00000002 POOL_EXTENTS_ANCHOR:00000000_0862E790
+000068 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:03
+00006A POOL_NUM_SAME_SIZE:01 POOL_TRACE_TABLE:00000000_0888C140

[1] Verifying free chain for pool: 3...
No errors were found while processing free chain.
Summary of analysis for Pool 3:
Number of cells: Unused: 0 Free: 0 Allocated: 0 Total Used: 0
No errors were found while processing this Pool.
The following is part two of the example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64).

Figure 195: Example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64) (Part 2 of 5)
The following is part three of the example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64).

Figure 196: Example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64) (Part 3 of 5)
The following is part four of the example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64).

<table>
<thead>
<tr>
<th>Data for pool 5.5:</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOLDATA: 00000008_08734500</td>
</tr>
<tr>
<td>+000008 CELL_SIZE:00000000 INPUT_CELL_SIZE:00000000</td>
</tr>
<tr>
<td>+000010 CELL_POOL_SIZE:00001000 CELL_POOL_NUM:00000000</td>
</tr>
<tr>
<td>+000010 POOL_LATCH_ADDR:00000000_08710000 POOL_EXTENTS:00000000</td>
</tr>
<tr>
<td>+000020 LAST_CELL:00000000_08711000 NEXT_CELL:00000000_08712000</td>
</tr>
<tr>
<td>+000040 Q_CONTROL_INFO:00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+000050 POOL_NUM_GET_TOTAL:00000000 00000000 POOL_NUM_FREE:00000000 00000000</td>
</tr>
<tr>
<td>+000050 POOL_INDEX_SAME_SIZE:05 POOL_INDEX_SIZE:05 POOL_NUM_SAME_SIZE:05</td>
</tr>
<tr>
<td>+000058 POOL_TRACE_TABLE:00000008_08A30260</td>
</tr>
</tbody>
</table>

There are no extents for this pool.

<table>
<thead>
<tr>
<th>Data for pool 6:</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOLDATA: 00000008_08734600</td>
</tr>
<tr>
<td>+000008 POOL_INDEX:00000000 INPUT_CELL_SIZE:00000000</td>
</tr>
<tr>
<td>+000010 CELL_POOL_SIZE:00002000 CELL_POOL_NUM:00000000</td>
</tr>
<tr>
<td>+000018 POOL_LATCH_ADDR:00000000_08710000 POOL_EXTENTS:00000000</td>
</tr>
<tr>
<td>+000028 LAST_CELL:00000000_0862DAD0 NEXT_CELL:00000000_0862DAD0</td>
</tr>
<tr>
<td>+000040 Q_CONTROL_INFO:00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+000050 POOL_NUM_GET_TOTAL:00000000 00000000 POOL_NUM_FREE:00000000 00000000</td>
</tr>
<tr>
<td>+000050 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:06 POOL_NUM_SAME_SIZE:01</td>
</tr>
<tr>
<td>+000058 POOL_TRACE_TABLE:00000008_08A76290</td>
</tr>
</tbody>
</table>

Heap Pool Extent Mapping

EXTENT: 00000000_0862C290
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_00000000

To display entire pool extent: IP LIST 000000080862C290 LEN(X'00002050') ASID(X'0021')

000000080862C2A0: Allocated storage cell. To display: IP LIST 000000080862C2A0 LEN(X'00000810') ASID(X'0021')

000000080862C2B0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000|................................|

000000080862CAB0: Free storage cell. To display: IP LIST 000000080862CAB0 LEN(X'00000810') ASID(X'0021')

000000080862D2C0: Allocated storage cell. To display: IP LIST 000000080862D2C0 LEN(X'00000810') ASID(X'0021')

000000080862D2D0: 00000008 082FA5A8 00000000 00000000 00000008 082FAE20 00000000 00000008|......vy........................|

Summary of analysis for Pool 6:
Number of cells:  Unused: 1 Free: 1 Allocated: 2 Total Used: 4
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

<table>
<thead>
<tr>
<th>Data for pool 7:</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOLDATA: 00000008_08734700</td>
</tr>
<tr>
<td>+000008 POOL_INDEX:00000000 INPUT_CELL_SIZE:00000000</td>
</tr>
<tr>
<td>+000010 CELL_SIZE:00000800 INPUT_COUNT:00000000</td>
</tr>
<tr>
<td>+000018 POOL_LATCH_ADDR:00000000_08710000 POOL_EXTENTS:00000000</td>
</tr>
<tr>
<td>+000028 LAST_CELL:00000000_08607F80 NEXT_CELL:00000000_08607F80</td>
</tr>
<tr>
<td>+000040 Q_CONTROL_INFO:00000000 00000000 00000000</td>
</tr>
<tr>
<td>+000050 POOL_NUM_GET_TOTAL:00000000 00000000 POOL_NUM_FREE:00000000 00000000</td>
</tr>
<tr>
<td>+000050 POOL_INDEX_SAME_SIZE:01 POOL_INDEX_SIZE:07 POOL_NUM_SAME_SIZE:01</td>
</tr>
<tr>
<td>+000058 POOL_TRACE_TABLE:00000008_08ABC2C0</td>
</tr>
</tbody>
</table>

Heap Pool Extent Mapping

EXTENT: 00000000_08606750
+000000 EYE_CATCHER:EX64 NEXT_EXTENT:00000000_00000000

To display entire pool extent: IP LIST 0000000808606750 LEN(X'00002050') ASID(X'0021')

0000000008606760: Allocated storage cell. To display: IP LIST 0000000008606760 LEN(X'00000C00') ASID(X'0021')

0000000008606770: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000|..............................|

0000000008606780: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000|..............................|

Summary of analysis for Pool 7:
Number of cells: Unused: 48 Free: 0 Allocated: 2 Total Used: 50
00000000 free cells were not accounted for.
No errors were found while processing this Pool.

<table>
<thead>
<tr>
<th>Data for pool 8:</th>
</tr>
</thead>
<tbody>
<tr>
<td>POOLDATA: 00000008_08734800</td>
</tr>
<tr>
<td>+000000 POOL_INDEX:00000000 INPUT_CELL_SIZE:00000000</td>
</tr>
<tr>
<td>+000010 CELL_SIZE:00000800 INPUT_COUNT:00000000</td>
</tr>
<tr>
<td>+000018 POOL_LATCH_ADDR:00000000_08710000 POOL_EXTENTS:00000000</td>
</tr>
<tr>
<td>+000020 LAST_CELL:00000000_08711000 NEXT_CELL:00000000_08712000</td>
</tr>
<tr>
<td>+000040 Q_CONTROL_INFO:00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>+000050 POOL_NUM_GET_TOTAL:00000000 00000000 POOL_NUM_FREE:00000000 00000000</td>
</tr>
<tr>
<td>+000050 POOL_INDEX_SAME_SIZE:05 POOL_INDEX_SIZE:05 POOL_NUM_SAME_SIZE:05</td>
</tr>
<tr>
<td>+000058 POOL_TRACE_TABLE:00000008_08A30260</td>
</tr>
</tbody>
</table>

Figure 197: Example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64) (Part 4 of 5)
The following is part five of the example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64).

Figure 198: Example formatted detailed heap pool report from LEDATA VERBEXIT (AMODE 64) (Part 5 of 5)

Heap pool report sections of the LEDATA output

As Table 56 on page 390 shows, the heap pool report provides information about the following items:

- Each cell pool.
- The free chain associated with every qpcb pool data area, and all the free and allocated cells in the extent chain.
- Errors found when the cells are validated.
### Table 56: Contents of the heap pool report sections of the LEDATA output (AMODE 64)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[1] Free Chain Validation</strong></td>
<td>Within each cell pool, Language Environment keeps track of unallocated cells by chaining them together. The LEDATA HEAP option validates the free chain within each cell pool. It verifies that the cell pointer is within a valid extent and that the cell pool number is valid. If the formatter finds a problem, it will place an error message describing the problem directly after the formatted line of the cell that failed validation.</td>
</tr>
<tr>
<td><strong>[2] Heap Pool Extent Mapping Report</strong></td>
<td>The LEDATA HEAP option produces a report that lists all of the cells within each pool extent, and identifies the cells as either allocated or freed. For each allocated cell, the contents of the first 'X'20' bytes of the area are displayed to identify the reason for the storage allocation. The formatter validates if the cell pool number in header is correct.</td>
</tr>
</tbody>
</table>

### Understanding the heap pools trace LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates a detailed heap pools trace report when the HPT option is used (see Figure 199 on page 390. The argument value is the ID of the pool to be formatted in the report. Table 57 on page 394 explains the contents of each section of the report.

---

```
HPT(3)
******************************************************************************
64 BIT LANGUAGE ENVIRONMENT DATA
******************************************************************************
Language Environment Product 04 V01 R0A.00

[1] HEAPPOOL64 Trace Table

Type: FREE  Cell Address: 00000001086588E0  Cpuid: 01  Tcb: 008D7820

CALL NAME          CALL ADDRESS       CALL OFFSET
GetStorage::~GetStorage()  0000000025B001B0   00000056
foo6()              0000000025B00348   0000006A
foo7()              0000000025B003D8   00000010
foo6()              0000000025B00410   00000010
foo5()              0000000025B00448   00000010
foo4()              0000000025B00480   00000010
foo3()              0000000025B004B8   00000010
foo2()              0000000025B004F0   00000010
foo1()              0000000025B00528   00000000

thread

Type: FREE  Cell Address: 0000000108658970  Cpuid: 01  Tcb: 008D7820

CALL NAME          CALL ADDRESS       CALL OFFSET
GetStorage::~GetStorage()  0000000025B001B0   00000056
foo9()              0000000025B00260   0000006E
foo8()              0000000025B00348   00000046
foo7()              0000000025B003D8   00000010
foo6()              0000000025B00410   00000010
foo5()              0000000025B00448   00000010
foo4()              0000000025B00480   00000010
foo3()              0000000025B004B8   00000010
foo2()              0000000025B004F0   00000010
foo1()              0000000025B00528   00000000

Figure 199: Example of formatted detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64) (Part 1 of 5)
```
The following is part two of the example of formatted detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64).

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>2008/03/14 18:20:40.239873</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>GET</td>
</tr>
<tr>
<td>Cell Address</td>
<td>0000000108658970</td>
</tr>
<tr>
<td>Cpuid</td>
<td>01</td>
</tr>
<tr>
<td>Tcb</td>
<td>008D7820</td>
</tr>
<tr>
<td>CALL NAME</td>
<td>GetStorage:GetStorage(int)</td>
</tr>
<tr>
<td>CALL ADDRESS</td>
<td>0000000025B00118</td>
</tr>
<tr>
<td>CALL OFFSET</td>
<td>00000058</td>
</tr>
<tr>
<td>foo9()</td>
<td>0000000025B00260</td>
</tr>
<tr>
<td>foo8()</td>
<td>0000000025B00348</td>
</tr>
<tr>
<td>foo7()</td>
<td>0000000025B00330</td>
</tr>
<tr>
<td>foo6()</td>
<td>0000000025B00410</td>
</tr>
<tr>
<td>foo5()</td>
<td>0000000025B00448</td>
</tr>
<tr>
<td>foo4()</td>
<td>0000000025B00480</td>
</tr>
<tr>
<td>foo3()</td>
<td>0000000025B00488</td>
</tr>
<tr>
<td>foo2()</td>
<td>0000000025B004F0</td>
</tr>
<tr>
<td>foo1()</td>
<td>0000000025B00528</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>2008/03/14 18:20:40.238024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>FREE</td>
</tr>
<tr>
<td>Cell Address</td>
<td>00000001086588E0</td>
</tr>
<tr>
<td>Cpuid</td>
<td>01</td>
</tr>
<tr>
<td>Tcb</td>
<td>008D7AD0</td>
</tr>
<tr>
<td>CALL NAME</td>
<td>GetStorage::~GetStorage()</td>
</tr>
<tr>
<td>CALL ADDRESS</td>
<td>0000000025B001B0</td>
</tr>
<tr>
<td>CALL OFFSET</td>
<td>00000056</td>
</tr>
<tr>
<td>foo8()</td>
<td>0000000025B00348</td>
</tr>
<tr>
<td>foo7()</td>
<td>0000000025B00330</td>
</tr>
<tr>
<td>foo6()</td>
<td>0000000025B00410</td>
</tr>
<tr>
<td>foo5()</td>
<td>0000000025B00448</td>
</tr>
<tr>
<td>foo4()</td>
<td>0000000025B00480</td>
</tr>
<tr>
<td>foo3()</td>
<td>0000000025B00488</td>
</tr>
<tr>
<td>foo2()</td>
<td>0000000025B004F0</td>
</tr>
<tr>
<td>foo1()</td>
<td>0000000025B00528</td>
</tr>
<tr>
<td>thread</td>
<td>0000000025B005CB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>2008/03/14 18:20:40.238021</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>FREE</td>
</tr>
<tr>
<td>Cell Address</td>
<td>0000000108658970</td>
</tr>
<tr>
<td>Cpuid</td>
<td>01</td>
</tr>
<tr>
<td>Tcb</td>
<td>008D7AD0</td>
</tr>
<tr>
<td>CALL NAME</td>
<td>GetStorage::~GetStorage()</td>
</tr>
<tr>
<td>CALL ADDRESS</td>
<td>0000000025B001B0</td>
</tr>
<tr>
<td>CALL OFFSET</td>
<td>00000056</td>
</tr>
<tr>
<td>foo9()</td>
<td>0000000025B00260</td>
</tr>
<tr>
<td>foo8()</td>
<td>0000000025B00348</td>
</tr>
<tr>
<td>foo7()</td>
<td>0000000025B00330</td>
</tr>
<tr>
<td>foo6()</td>
<td>0000000025B00410</td>
</tr>
<tr>
<td>foo5()</td>
<td>0000000025B00448</td>
</tr>
<tr>
<td>foo4()</td>
<td>0000000025B00480</td>
</tr>
<tr>
<td>foo3()</td>
<td>0000000025B00488</td>
</tr>
<tr>
<td>foo2()</td>
<td>0000000025B004F0</td>
</tr>
<tr>
<td>foo1()</td>
<td>0000000025B00528</td>
</tr>
<tr>
<td>thread</td>
<td>0000000025B005CB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>2008/03/14 18:20:40.238016</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>GET</td>
</tr>
<tr>
<td>Cell Address</td>
<td>0000000108658970</td>
</tr>
<tr>
<td>Cpuid</td>
<td>01</td>
</tr>
<tr>
<td>Tcb</td>
<td>008D7AD0</td>
</tr>
<tr>
<td>CALL NAME</td>
<td>GetStorage:GetStorage(int)</td>
</tr>
<tr>
<td>CALL ADDRESS</td>
<td>0000000025B00118</td>
</tr>
<tr>
<td>CALL OFFSET</td>
<td>00000058</td>
</tr>
<tr>
<td>foo9()</td>
<td>0000000025B00260</td>
</tr>
<tr>
<td>foo8()</td>
<td>0000000025B00348</td>
</tr>
<tr>
<td>foo7()</td>
<td>0000000025B00330</td>
</tr>
<tr>
<td>foo6()</td>
<td>0000000025B00410</td>
</tr>
<tr>
<td>foo5()</td>
<td>0000000025B00448</td>
</tr>
<tr>
<td>foo4()</td>
<td>0000000025B00480</td>
</tr>
<tr>
<td>foo3()</td>
<td>0000000025B00488</td>
</tr>
<tr>
<td>foo2()</td>
<td>0000000025B004F0</td>
</tr>
<tr>
<td>foo1()</td>
<td>0000000025B00528</td>
</tr>
</tbody>
</table>

Figure 200: Example of formatted detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64) (Part 2 of 5)
The following is part three of the example of formatted detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64).

Figure 201: Example of formatted detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64) (Part 3 of 5)
The following is part four of the example of formatted detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64).

Figure 202: Example of formatted detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64) (Part 4 of 5)
The following is part five of the example of formatted detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64).

Figure 203: Example of formatted detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64)
(Part 5 of 5)

Table 57: Contents of a detailed heap pools trace report from LEDATA VERBEXIT (AMODE 64)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] Trace Header</td>
<td>HEAPPOOLS64 trace header information.</td>
</tr>
<tr>
<td>[2] Pool Information</td>
<td>Includes the number of the pool (pool ID) that is currently being formatted, the ASID, the number of entries formatted, and the total number of entries taken. The trace wraps for each pool ID after a specific number of entries. The number of entries is controlled by the HEAPCHK runtime option.</td>
</tr>
<tr>
<td>[3] Timestamp</td>
<td>The time this trace entry was taken. The trace entries are formatted in reverse order (most recent trace entry first).</td>
</tr>
<tr>
<td>[4] Trace Table Entry</td>
<td>The individual trace entry, which contains:</td>
</tr>
<tr>
<td>contents</td>
<td>• The TYPE - GET or FREE.</td>
</tr>
<tr>
<td></td>
<td>• The Cell within the pool being acted upon.</td>
</tr>
<tr>
<td></td>
<td>• The CPU and TCB that requested or freed the cell.</td>
</tr>
<tr>
<td></td>
<td>• A traceback at the time of the request. The number of entries in this traceback is limited by the HEAPCHK runtime option.</td>
</tr>
</tbody>
</table>

Understanding the C/C++-specific LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of C/C++-specific control blocks from a system dump when the COMP(C), COMP(ALL), or ALL parameter is specified and C/C++ is active in the dump. Figure 204 on page 395 illustrates the C/C++-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option when running the program CELQSAMP Figure 162 on page 340. “C/C++-specific sections of the LEDATA output” on page 404 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 204: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 10)
The following is part two of the example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64)

Figure 205: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 2 of 10)
The following is part three of the example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64)

Figure 206: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 3 of 10)
Figure 207: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 4 of 10)
The following is part five of the example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64)

File name: memory.data

<table>
<thead>
<tr>
<th>FCB</th>
<th>BUFPTR:00000000_2135C0F8</th>
<th>COUNTIN:00000000 00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>BUFPTR:00000000_2135B0E3</td>
<td>COUNTIN:00000000 00000000</td>
</tr>
<tr>
<td>+000100</td>
<td>COUNTOUT:00000000_2135A998</td>
<td>LENGTH:00000000 00000008</td>
</tr>
<tr>
<td>+000010</td>
<td>COUNTOUT:00000000_2135A998</td>
<td>LENGTH:00000000 00000008</td>
</tr>
<tr>
<td>+000040</td>
<td>BUFPTR:00000000_2135A998</td>
<td>LENGTH:00000000 00000008</td>
</tr>
<tr>
<td>+000068</td>
<td>CHILD:00000000 00000000</td>
<td>DNAME:...... FD:FFFFFFF</td>
</tr>
<tr>
<td>+000070</td>
<td>DEVTYP0:99 FCBTYPP:0007C</td>
<td>FSCE:00000000_2131F7B0</td>
</tr>
<tr>
<td>+000080</td>
<td>UNGETBUF:00000000_2135A888</td>
<td>REPOS:00000000_2131F7B0</td>
</tr>
<tr>
<td>+000090</td>
<td>GETPOS:00000000_2131F850</td>
<td>CLOS0:00000000_2131F7B0</td>
</tr>
<tr>
<td>+000100</td>
<td>FLUSH:00000000_2131F850</td>
<td>CLOS0:00000000_2131F7B0</td>
</tr>
<tr>
<td>+000110</td>
<td>USERBUF:00000000_2135A6A0</td>
<td>LRECL:00000000 00000000</td>
</tr>
<tr>
<td>+000120</td>
<td>BLKSIZE:00000000 00000000</td>
<td>REPBS:00000000 00000000</td>
</tr>
<tr>
<td>+000130</td>
<td>MEMO:00000000_2135A678</td>
<td>MARKER1:AFCB</td>
</tr>
</tbody>
</table>

File name: myfile.data

<table>
<thead>
<tr>
<th>FCB</th>
<th>BUFPTR:00000000_2135C0F8</th>
<th>COUNTIN:00000000 00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000000</td>
<td>BUFPTR:00000000_2135B0E3</td>
<td>COUNTIN:00000000 00000000</td>
</tr>
<tr>
<td>+000100</td>
<td>COUNTOUT:00000000_2135A998</td>
<td>LENGTH:00000000 00000008</td>
</tr>
<tr>
<td>+000010</td>
<td>COUNTOUT:00000000_2135A998</td>
<td>LENGTH:00000000 00000008</td>
</tr>
<tr>
<td>+000040</td>
<td>BUFPTR:00000000_2135A998</td>
<td>LENGTH:00000000 00000008</td>
</tr>
<tr>
<td>+000068</td>
<td>CHILD:00000000 00000000</td>
<td>DNAME:...... FD:00000000</td>
</tr>
<tr>
<td>+000070</td>
<td>DEVTYP0:99 FCBTYPP:0007C</td>
<td>FSCE:00000000_2131F7B0</td>
</tr>
<tr>
<td>+000080</td>
<td>UNGETBUF:00000000_2135A888</td>
<td>REPOS:00000000_2131F7B0</td>
</tr>
<tr>
<td>+000090</td>
<td>GETPOS:00000000_2131F850</td>
<td>CLOS0:00000000_2131F7B0</td>
</tr>
<tr>
<td>+000100</td>
<td>FLUSH:00000000_2131F850</td>
<td>CLOS0:00000000_2131F7B0</td>
</tr>
<tr>
<td>+000110</td>
<td>USERBUF:00000000_2135A6A0</td>
<td>LRECL:00000000 00000000</td>
</tr>
<tr>
<td>+000120</td>
<td>BLKSIZE:00000000 00000000</td>
<td>REPBS:00000000 00000000</td>
</tr>
<tr>
<td>+000130</td>
<td>MEMO:00000000_2135A678</td>
<td>MARKER1:AFCB</td>
</tr>
</tbody>
</table>

Figure 208: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 5 of 10)
The following is part six of the example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64)

| HFSF: 00000000_2135A888 |
| +000000 HFSF_EYE:HFSF READ:00000000_2131F210 WRITE:00000000_2131F7A0 |
| +000018 REPOS:00000000_2131F870 GETPOS:00000000_2131F860 |
| +000028 FLUSH:00000000_2131F580 READBUFSIZE:00000000 00000000 |
| +000038 OPENFLAG:00000049 FLAG1:00000000 HFSF_ST_MODE:030001A4 |
| +000048 HFSF_LAST_FSTAT:43281E34 A142E3FC |
| FFIL: 00000001_0000A2B0 |
| +000000 MARKER1:AFCB __FP:00000001_0000A2D8 |
| +000010 MARKER2:AFCB ACB FCBMUTEX:00000001_08FC7290 |
| +000020 THREADID:2109B260 00000000 |

File name: DD:SYSIN

| FCB: 00000001_0000A2D8 |
| +000000 BUFPTR:00000000_00000000 COUNTIN:00000000 00000000 |
| +000010 COUNTOUT:00000000 00000000 READFUNC:00000000_2131FEC0 |
| +000020 WRITEFUNC:00000000_2131F200 FLAGS1:8000 DEPTH:0000 |
| +000030 NAME:00000001_0000A5D0 _LENGTH:00000000 0000000B |
| +000040 _BUFSIZE:00000000 00000048 MEMBER:........ NEXT:00000001_00009F70 |
| +000058 PREV:00000000_2135A6A0 PARENT:00000001_0000A2D8 |
| +000068 CHILD:00000000 00000000 DDNAME:SYSIN FD:FFFFFFFF |
| +00007D DEVTYPE:06 FCBTYPE:0041 FSCE:00000001_0000A4C0 |
| +000088 UNGETBUF:00000001_0000A4C0 REPOS:00000000_213206C0 |
| +000098 GETPOS:00000000_2131FC60 CLOSE:00000000_2131FD80 |
| +0000A8 FLUSH:00000000_2131FEB0 UTILITY:00000000_2131FEA0 |
| +0000B8 USERBUF:00000000_00000000 LRECL:00000000 00000000 |
| +0000C8 BLKSIZE:00000000 00001800 REALBUFPTR:00000000_00000000 |
| +0000D8 UNGETCOUNT:00000000 00000000 BUFSIZE:00000000 00001801 |
| +0000E8 BUF:00000000_00000000 CURSOR:00000000_00000000 |
| +0000F8 ENDOFDATA:00000000_00000000 SAVEDBUF:00000000_00000000 |
| +000108 REALCOUNTIN:00000000 00000000 |
| +000110 REALCOUNTOUT:00000000 00000000 |
| +000120 SAVEVMAJOR:00000000 00000000 POSMIGN:00000000 00000000 |
| +000130 SAVEMINOR:00000000 00000000 STATE:0000 SAVESTATE:0000 |
| +000140 EXITFTELL:00000000_2131FC20 EXITUNGETC:00000000_00000000 |
| +000150 DBCSSTART:00000000_2131FC50 DCBSRG1:40 DCBEODAD:000000 |
| +000160 INTERCEPT:00000000_2131FD40 FLAGS2:01800000 00000000 |
| +000170 DBCSSTATE:0000 FCB_CPCB:00000000_2135A638 |
| +000180 LLPOSMIGN:00000000 00000000 |
| +000190 LLSAVEVMAJOR:00000000 00000000 |
| +0001B0 LLSAVEVMINOR:00000000 00000000 |

| OSNS: 00000001_0000A4AC |
| +000000 OSNS_EYE:OSNS READ:00000000_2131F70 WRITE:00000000_2131F200 |
| +000010 REPOS:00000000_2131F870 GETPOS:00000000_2131F860 |
| +000020 FLUSH:00000000_2131F580 UTILITIES:00000000 2131F8A0 |
| +000030 UTILITY:00000000_2131F8A0 EXTIFTELL:00000000 2131F8A0 |
| +000040 EXITFTELL:00000000_2131FD20 EXITUNGETC:00000000_00000000 |
| +000050 NEWLINEPTR:00000000_00000000 RECLENGTH:00000000 00018000 |
| +000060 FLAGS:01800000 |

| OS10: 00000000_2135A5CB |
| +000000 OS10_EYE:OS10 CURRBUFF:00000000 00000000 DCB:00000000_00000000 |
| +00000C JFBC:000007A6 CURRMDUM:00000000 00000000 DOSTK:00000000 |
| +00000E READMAX:00000000 00000000 CURBLKSIZE:00000000 00000000 |
| +000014 OS10_ACCESS_METHOD:02 OS10_ACCESS_METHOD:02 |
| +00001E OS10_ACCESS_METHOD:02 OS10_ACCESS_METHOD:02 |
| +000020 OS10_ACCESS_METHOD:02 OS10_ACCESS_METHOD:02 |
| +000024 OS10_ACCESS_METHOD:02 OS10_ACCESS_METHOD:02 |
| +000034 OS10_ACCESS_METHOD:02 OS10_ACCESS_METHOD:02 |
| +000048 OS10_ACCESS_METHOD:02 OS10_ACCESS_METHOD:02 |
| +000050 OS10_ACCESS_METHOD:02 OS10_ACCESS_METHOD:02 |
| +000060 OS10_ACCESS_METHOD:02 OS10_ACCESS_METHOD:02 |
| +000064 OS10_ACCESS_METHOD:02 OS10_ACCESS_METHOD:02 |

| DCB: 00000000_00007A00 |
| +000000 DCBRELAD:2135A63B DCFAD:00000000 00000000 DCH:00000000_00000000 |
| +00000C DCH:00000000_00000000 DCFAD:00000000 00000000 DCH:00000000_00000000 |
| +000020 DCBREL:00000000 00000000 DCBRECFM:CD DCBNAME:u...... DCBNAME:u...... |
| +000033 DCBNAME:0000_00000000 DCBRECFM:CD DCBNAME:0000_00000000 |
| +000052 DCBRECFM:CD DCBNAME:0000_00000000 DCBRECFM:CD DCBNAME:0000_00000000 |
| +000056 DCBRECFM:CD DCBNAME:0000_00000000 DCBRECFM:CD DCBNAME:0000_00000000 |

Figure 209: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 6 of 10)
The following is part seven of the example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64)

Figure 210: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 7 of 10)
The following is part eight of the example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64)

File name: DD:SYSPRINT

Figure 211: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 8 of 10)
The following is part nine of the example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64)

```
+0001C0 LLPOSMAJOR:00000000 00000000
+0001C8 LLSAVEMAJOR:00000000 00000000
+0001D0 LLPOSMINOR:00000000 00000000
+0001D8 LLSAVEMINOR:00000000 00000000

OSNS: 00000001_00009DF0
+000200 OSNS_EYE:OSNS  READ:00000000_2131F210  WRITE:00000000_2131FDC0
+000210 REPOS:00000000_2131FDE0  GETPOS:00000000_2131FDB0
+000220 CLOSE:00000000_2131FDB0  FLUSH:00000000_2131FDE0
+000230 UTILITY:00000000_2131F780  EXITFTELL:00000000_2131FC20
+000240 NEWLINEPTR:00000000_2135A509  RECLENGTH:00000000 00000005
+000250 FLAGS:00000000_2135A509

OSIO: 00000000_2135A3D0
+000260 OSIO_EYE:OSIO  DCBW:00000000_2135A3D0  DCBRU:00000000
+000270 JFCB:000077F0  CURRMBUF:00007048  CURRMBUF:00000000
+000280 JFCBDSNM:HEALY.CELQSAMP.JOB24799.D0000102.
+000290 JFCBELNM:  JFCEWLSQ:00000000  JFCBHLSQ:00000000  JFCBSQ:00000000
+0002A0 JFCEWLSQ:00000000  JFCBHLSQ:00000000  JFCBSQ:00000000

DCB: 00000000_00007048
+000300 DCBBUFNO:00000001  DCBSRQ1:40  DCBDNAM:00000000  DCBRECFM:54
+000310 DCBEXLSA:00000000  DCBEXLSA:00000000  DCBEXLSA:00000000
+000320 DCBEXLSA:00000000  DCBExLSA:00000000  DCBExLSA:00000000
+000330 DCBEXLSA:00000000  DCBExLSA:00000000  DCBExLSA:00000000
+000340 DCBEXLSA:00000000  DCBExLSA:00000000  DCBExLSA:00000000
+000350 DCBEXLSA:00000000  DCBExLSA:00000000  DCBExLSA:00000000

DCBE: 00000000_2135A440
+000360 DCBCB:00007048  DCBBURU:00000000  DCBFR:00000000  DCBFR:00000000
+000370 DCBFR:00000000  DCBFR:00000000  DCBFR:00000000
+000380 DCBFR:00000000  DCBFR:00000000  DCBFR:00000000
+000390 DCBFR:00000000  DCBFR:00000000  DCBFR:00000000
+0003A0 DCBFR:00000000  DCBFR:00000000  DCBFR:00000000
+0003B0 DCBFR:00000000  DCBFR:00000000  DCBFR:00000000

JFCE: 00000000_2135A440
+0003C0 JFCEFORM:00000000  JFCBFORM:00000000  JFCBFORM:00000000
+0003D0 JFCBFORM:00000000  JFCBFORM:00000000  JFCBFORM:00000000
+0003E0 JFCBFORM:00000000  JFCBFORM:00000000  JFCBFORM:00000000
+0003F0 JFCBFORM:00000000  JFCBFORM:00000000  JFCBFORM:00000000
+000400 JFCBFORM:00000000  JFCBFORM:00000000  JFCBFORM:00000000

MBUF: 00000000_000078A8
+000410 MBUF:00000000_000078A8  BUFFER:2135A480_ CHECKRESULT:00000000
+000420 BLKSIZE:00000000  BLKSIZE:00000000  BLKSIZE:00000000
+000430 BLKSIZE:00000000  BLKSIZE:00000000  BLKSIZE:00000000

Dummy FCB encountered at location 00000001_00008A08
```

Figure 212: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 9 of 10)
The following is part ten of the example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64)

```
AMRC: 00000000_2135A278
+000000 CODE:00000000 RBA:0000000000 LAST_OP:0000000098
+000000C FILL_LEN:00000000 MSG_LEN:00000000
+000014 STRI:............................................................
+000050 STRI_CONT:............................................................
+000060 PARM0R0:00000000 PARM1R:00000000
+00006C STR2:................................................................
+0000DC RPLFBWD:00000000 XRBA:00000000 0000000000
+0000F0 AMRC_NOSEEK_TO_SEEK:00
AMRC2: 00000000_2135A380
+000000 __ERROR2:00000000 00000000 __FILEPTR:00000000_

File name: memory.data
```

Figure 213: Example of formatted C/C++ output from LEDATA VERBEXIT (AMODE 64) (Part 10 of 10)

**C/C++-specific sections of the LEDATA output**

Table 58 on page 404 describes the contents of the LEDATA output that is specific to C/C++.

| Table 58: Contents of C/C++-specific sections of the LEDATA output (AMODE 64) |
|--------------------------------|----------------------------------|
| **Section Number and Heading** | **Contents**                     |
| [1] CGEN                      | Formats the C/C++-specific portion of the Language Environment common anchor area (CAA). |
| [2] CGENE                     | Formats the extension to the C/C++-specific portion of the Language Environment common anchor area (CAA). |
Table 58: Contents of C/C++-specific sections of the LEDATA output (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[7] File Control Blocks</strong></td>
<td>Formats the C/C++ file control block (FCB). The FCB and its related control blocks, which are listed below, represent the information needed by each open stream.</td>
</tr>
<tr>
<td>FFIL</td>
<td>Formats the header of the C/C++ file control block (FCB).</td>
</tr>
<tr>
<td>FSCE</td>
<td>The file specific category extension control block, which represents the specific type of IO being performed. The following FSCEs may be formatted; other FSCEs will be displayed using a generic overlay.</td>
</tr>
<tr>
<td>HFSF</td>
<td>UNIX file system file</td>
</tr>
<tr>
<td>HSPF</td>
<td>Hiper-Space file</td>
</tr>
<tr>
<td>INTC</td>
<td>Intercept file</td>
</tr>
<tr>
<td>MEMF</td>
<td>Memory file</td>
</tr>
<tr>
<td>OSNS</td>
<td>OS no seek</td>
</tr>
<tr>
<td>OSFS</td>
<td>OS fixed text</td>
</tr>
<tr>
<td>OSVF</td>
<td>OS variable text</td>
</tr>
<tr>
<td>OSUT</td>
<td>OS undefined format text</td>
</tr>
<tr>
<td>TDQF</td>
<td>CICS Transient Data Queue file</td>
</tr>
<tr>
<td>TERM</td>
<td>Terminal file</td>
</tr>
<tr>
<td>VSAM</td>
<td>VSAM file</td>
</tr>
<tr>
<td>OSIO</td>
<td>The OS IO interface control block.</td>
</tr>
<tr>
<td>OSIOE</td>
<td>The OS IO extended interface control block.</td>
</tr>
<tr>
<td>DCB</td>
<td>The data control block; for more information, see z/OS DFSMS Macro Instructions for Data Sets.</td>
</tr>
<tr>
<td>DCBE</td>
<td>The data control block extension; for more information, see z/OS DFSMS Macro Instructions for Data Sets.</td>
</tr>
<tr>
<td>JFCB</td>
<td>The job file control block (JFCB); for more information, see z/OS MVS Data Areas in the z/OS Internet library (<a href="http://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary">www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary</a>).</td>
</tr>
<tr>
<td>JFCBX</td>
<td>The job file control block extension (JFCBX).</td>
</tr>
<tr>
<td>MBUF</td>
<td>The message buffer control block (MBUF).</td>
</tr>
</tbody>
</table>

| **[8] Memory File Control Blocks** | Formats the C/C++ memory file control block (MFCB). |
Understanding the PL/I-specific LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of PL/I-specific control blocks from a system dump when the ALL parameter is specified and PL/I is active in the dump. Figure 214 on page 406 illustrates the PL/I-specific output produced. The system dump being formatted was obtained by specifying the TERMTHDACT(UADUMP) runtime option. “PL/I-specific sections of the LEDATA output” on page 411 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 214: Example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 6)
The following is part two of the example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64).

Figure 215: Example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64) (Part 2 of 6)
The following is part three of the example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64).

+00009B SLD_ONLOC:00000050 08601388 02040000 00000000
+0000CC RES_EBP:00000000_00000000 RES_EIP:00000000_00000000
+0000DC TRCBK_EBP:00000000_00000000 TRCBK_EIP:00000000_00000000
+0000FC RETRY_EBP:00000000_00000000 RETRY_EIP:00000000_00000000
+00010C EB_P_HND:00000000_00000000 PLISRXT_RC:00000000
+000118 UNDEF_SUBC1:00000000_00000000 UNDEF_SUBC2:00000000
+000124 TOKEN:00000000_00000000 00000000 00000000
+000134 AMODE_SWC_PTR:00000000_00000000
+000140 ONLINE:00000000
+000158 --------------------- start of data for file 1 ---------------------

FILENAME:VSAMA
FNAME:  00000050_0860343C
NAMELEN:0005
PFO:  00000050_0860340C
ANCHOR:00000050_08603434      DECLARED:00840801
INVALIDS:00600602       NAMEPTR:00000050_0860343C
ENVPTR:00000000_0F1007F0      INT_TAG:00000000_00000000
ATTRS:KEYED EXT SEQ RECORD
INVLD:PRINT EXCL DIR TRANS STREAM

SHAD_FCO:  00000050_08603310
SELF:00000050_08603310  CHAIN:00000050_08602830
ANCESTOR:00000050_08603310    INV_STMT_METH:00000050_00123278
STMT_ERR_METH:00000050_001232A8
DIAGNOSE_METH:00000050_00123268 DONE_METH:00000050_001232A8
OPEN_METH:00000050_00123258 CLOSE_METH:00000050_001232A8
LOCATE_METH:00000050_00123318 WRTE_METH:00000050_001232E8
INVALIDS:00600602       NAMEPTR:00000050_0860343C
STMT_ERR_METH:00000050_001219F8
DIAGNOSE_METH:00000050_00123268 DONE_METH:00000050_001232A8
OPEN_METH:00000050_00123258 CLOSE_METH:00000050_001232A8
LOCATE_METH:00000050_00123318 WRTE_METH:00000050_001232E8
INVALIDS:00600602       NAMEPTR:00000050_0860343C
STMT_ERR_METH:00000050_001219F8
DIAGNOSE_METH:00000050_00123268 DONE_METH:00000050_001232A8
OPEN_METH:00000050_00123258 CLOSE_METH:00000050_001232A8
LOCATE_METH:00000050_00123318 WRTE_METH:00000050_001232E8
INVALIDS:00600602       NAMEPTR:00000050_0860343C
STMT_ERR_METH:00000050_001219F8
DIAGNOSE_METH:00000050_00123268 DONE_METH:00000050_001232A8
OPEN_METH:00000050_00123258 CLOSE_METH:00000050_001232A8
LOCATE_METH:00000050_00123318 WRTE_METH:00000050_001232E8
INVALIDS:00600602       NAMEPTR:00000050_0860343C
STMT_ERR_METH:00000050_001219F8
DIAGNOSE_METH:00000050_00123268 DONE_METH:00000050_001232A8
OPEN_METH:00000050_00123258 CLOSE_METH:00000050_001232A8
LOCATE_METH:00000050_00123318 WRTE_METH:00000050_001232E8
INVALIDS:00600602       NAMEPTR:00000050_0860343C
STMT_ERR_METH:00000050_001219F8
DIAGNOSE_METH:00000050_00123268 DONE_METH:00000050_001232A8
OPEN_METH:00000050_00123258 CLOSE_METH:00000050_001232A8
LOCATE_METH:00000050_00123318 WRTE_METH:00000050_001232E8
INVALIDS:00600602       NAMEPTR:00000050_0860343C
STMT_ERR_METH:00000050_001219F8
DIAGNOSE_METH:00000050_00123268 DONE_METH:00000050_001232A8
OPEN_METH:00000050_00123258 CLOSE_METH:00000050_001232A8
LOCATE_METH:00000050_00123318 WRTE_METH:00000050_001232E8
INVALIDS:00600602       NAMEPTR:00000050_0860343C
STMT_ERR_METH:00000050_001219F8
DIAGNOSE_METH:00000050_00123268 DONE_METH:00000050_001232A8
OPEN_METH:00000050_00123258 CLOSE_METH:00000050_001232A8
LOCATE_METH:00000050_00123318 WRTE_METH:00000050_001232E8
INVALIDS:00600602       NAMEPTR:00000050_0860343C
SHADOW_PFO:00000000_0860340C
INIT_PFO:00000000_00000000 INIT_PFO_ANC:00000000_00000000

Figure 216: Example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64) (Part 3 of 6)
The following is part four of the example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64).

Figure 217: Example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64) (Part 4 of 6)
The following is part five of the example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64).

```
+00028  DIAGNOSE_METH:00000050_00123268  DONE_METH:00000050_001232A8
+00038  OPEN_METH:00000050_01121358  CLOSE_METH:00000050_011232A8
+00048  CONTROL_METH:00000050_01121358  LOCATE_METH:00000050_01123278
+00058  WRITE_METH:00000050_01121328  REWRITE_METH:00000050_01123278
+00068  DELETE_METH:00000050_01123278  WAIT_METH:00000050_011232A8
+00078  UNLOCK_METH:00000050_01123248  GET_METH:00000050_011232A8
+00088  PATHNAME:00000050_00122298  INIT_PFO:00000050_001232A8
+00098  WRITE_METH:00000050_00123248  READ_METH:00000050_00123278
+00108  CTRL_METH:00000050_00123358  QRYTYPE_METH:00000050_01143780
+00118  PUT_METH:00000050_00122298  GET_METH:00000050_001232A8
```

The following is part six of the example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64).

```
+00128  FLUSH_METH:00000050_001232B8   FINDUSE_METH:00000050_00123298
+00138  DELETE_METH:00000050_00123278  READ_METH:00000050_00123278
+00148  UNLOCK_METH:00000050_00123248  GET_METH:00000050_001232A8
+00158  PUT_METH:00000050_00122298  GET_METH:00000050_001232A8
```

Figure 218: Example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64) (Part 5 of 6)

Figure 219: Example of formatted PL/I output from LEDATA VERBEXIT (AMODE 64) (Part 6 of 6)
Table 59 on page 411 describes the contents of the LEDATA output that is specific to C/C++.

<table>
<thead>
<tr>
<th>Section Number and Heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] PCB</td>
<td>This section formats the Enterprise PL/I process-level control block (PCB).</td>
</tr>
<tr>
<td>[2] DYNLST</td>
<td>This section formats the Enterprise PL/I process-level dynamic allocation parameter list.</td>
</tr>
<tr>
<td>[3] TCA</td>
<td>This section formats the Enterprise PL/I task communication control block (TCA).</td>
</tr>
<tr>
<td>[4] FECB</td>
<td>This section formats the PL/I for MVS and VM fetch control block (FECB).</td>
</tr>
<tr>
<td>[5] OCA</td>
<td>This section formats the Enterprise PL/I ON communications control block (OCA).</td>
</tr>
<tr>
<td>[6] PFO</td>
<td>This section formats the Enterprise PL/I file object control block (PFO).</td>
</tr>
<tr>
<td>[7] SHAD_FCO</td>
<td>This section formats the Enterprise PL/I shadow file object control block (shadow FCO).</td>
</tr>
<tr>
<td>[8] FCO</td>
<td>This section formats the Enterprise PL/I file control block (FCO).</td>
</tr>
<tr>
<td>[9] SCB</td>
<td>This section formats the Enterprise PL/I stream I/O control block (SCB).</td>
</tr>
<tr>
<td>[10] ATTRS</td>
<td>This section formats the Enterprise PL/I file attribute map (ATTRS).</td>
</tr>
<tr>
<td>[12] DCB</td>
<td>This section formats the Enterprise PL/I data control block (DCB).</td>
</tr>
</tbody>
</table>

Understanding the AUTH LEDATA output

The Language Environment IPCS VERBEXIT LEDATA generates formatted output of Preinitialized Environments for Authorized Programs-specific control blocks from a system dump when the AUTH parameter is specified. Figure 220 on page 412 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

[1]  ALEC:  00000000_7F6F7000
+000000  ID:ALEC  Ascb:00FBBE00  Flags1:40000000
+000000  UseCount:00000001  ASATable@1:7F6E8414
+000104  ASATable@2:7F6E8754  ASATable@3:00000000
+000104  ASATable@4:00000000  MCallRtn:82993A80
+000204  UCallRtn:82999DB8  LatchSetTok:7F6C0B40 00000074
+000300  Alei:00000001_001053A0  Ales:00000001_00107CD0
+000400  StackCPID:7F6C3F00  AROTCB:008FF028  EnvTypeNum:00000000
+000500  WorkECB:808E6F10
+000550  AROTToken:0000006C 00000003 00000003 008FF028
+000600  WTPE:00000053 023B8240 00000000 00000000  ALELVT:00000001
+000700  SLELVT:00100140  FuncTable@:00000000_00000000
+000800  WorkQueue:00000000_00000000  ALESeqNum:00000000 00000027
+000900  ALESCount:00000000 00000001  SystemRtnCode:00000000
+000950  SystemRsnCode:00000000  SystemRtnCodeJr:00000000
+000A00  SystemRtnCodeJr:00000000  WorkerTCB:008D8E88
+000B00  SystemOCB@:00000000_00000000

[2] Load Module Control Blocks

Queue #: 0000000000000000

ALMI:  00000001_00100F40
+000000  ID:ALMI  ModuleSize:00000600  ModuleName:CELQDSNF
+000100  UseCount:00000000 00000001  LoadPoint:00000000 264E3000
+000200  EntryPoint:00000000 264E3001

ALMI:  00000001_00100B40
+000000  ID:ALMI  ModuleSize:0000F000  ModuleName:CDAEQED
+000100  UseCount:00000000 00000001  LoadPoint:00000000 26396000
+000200  EntryPoint:00000000 26396001

ALMI:  00000001_00100540
+000000  ID:ALMI  ModuleSize:000017AD  ModuleName:CEEMENU3
+000100  UseCount:00000000 00000000  LoadPoint:00000000 26386298
+000200  EntryPoint:00000000 26386298

Queue #: 0000000000000002

ALMI:  00000001_00101340
+000000  ID:ALMI  ModuleSize:00000450  ModuleName:EDCUCSNM
+000100  UseCount:00000000 00000001  LoadPoint:00000000 05F15640
+000200  EntryPoint:00000000 05F15640

Queue #: 0000000000000003

ALMI:  00000001_00100040
+000000  ID:ALMI  ModuleSize:00018800  ModuleName:CAEODPI
+000100  UseCount:00000000 00000001  LoadPoint:00000000 26414000
+000200  EntryPoint:00000000 26414001

ALMI:  00000001_00100340
+000000  ID:ALMI  ModuleSize:00002000  ModuleName:ALEMQ01
+000100  UseCount:00000000 00000003  LoadPoint:00000000 26384000
+000200  EntryPoint:00000000 26384001

Queue #: 0000000000000006

ALMI:  00000001_00100740
+000000  ID:ALMI  ModuleSize:00000400  ModuleName:CDIVZERO
+000100  UseCount:00000000 00000001  LoadPoint:00000000 26393000
+000200  EntryPoint:00000000 26393001

Queue #: 0000000000000007

ALMI:  00000001_00101140
+000000  ID:ALMI  ModuleSize:00000655  ModuleName:CEL4CTBL
+000100  UseCount:00000000 00000001  LoadPoint:00000000 264E7D58
+000200  EntryPoint:00000000 264E7D58

[3] User Managed Control Blocks

Figure 220: Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 1 of 4)
The following is part two of the example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64).

<table>
<thead>
<tr>
<th>Routine Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue #: 0000000000000010</td>
</tr>
<tr>
<td>Routine: CDIVZERO</td>
</tr>
<tr>
<td>ALRI: 00000001_00109040</td>
</tr>
<tr>
<td>+000000 ID:ALRI Flags:00000000 InstanceNum:00000000 00000026</td>
</tr>
<tr>
<td>+000100 NEXT:00000000_00000000 ALEC:00000000_00000000</td>
</tr>
<tr>
<td>+000200 AleiAddress:00000001_001053A0 AleiInstanceNum:00000000_00000025</td>
</tr>
<tr>
<td>+000400 RoutineNameLen:00000000 00000008 RoutineAddr:00000001_001055B8</td>
</tr>
<tr>
<td>+000600 EnqFlags:40000000 FuncEnv:00000000_00000000</td>
</tr>
<tr>
<td>+000800 FuncEntry:00000000_2639317B MasterAlri:00000000_00000000</td>
</tr>
<tr>
<td>+000A00 Ales:00000000_00000000 LUAIri:00000000_00000000</td>
</tr>
<tr>
<td>+000C00 EnclaveSeq#:00000001</td>
</tr>
<tr>
<td>+000E00 NextEnvAlri:00000000_00000000</td>
</tr>
<tr>
<td>Queue #: 0000000000000015</td>
</tr>
<tr>
<td>Routine: CDIVZERO</td>
</tr>
<tr>
<td>ALRI: 00000001_00109040</td>
</tr>
<tr>
<td>+000000 ID:ALRI Flags:00000000 InstanceNum:00000000 00000026</td>
</tr>
<tr>
<td>+000100 NEXT:00000000_00000000 ALEC:00000000_00000000</td>
</tr>
<tr>
<td>+000200 AleiAddress:00000001_001053A0 AleiInstanceNum:00000000_00000025</td>
</tr>
<tr>
<td>+000400 RoutineNameLen:00000000 00000008 RoutineAddr:00000001_001055B8</td>
</tr>
<tr>
<td>+000600 DlKeylen:00000000_00000000 ParmLen:00000000</td>
</tr>
<tr>
<td>+000800 EnqFlags:40000000 FuncEnv:00000000_00000000</td>
</tr>
<tr>
<td>+000A00 FuncEntry:00000000_2639317B MasterAlri:00000000_00000000</td>
</tr>
<tr>
<td>+000C00 Ales:00000000_00000000 LUAIri:00000000_00000000</td>
</tr>
<tr>
<td>+000E00 EnclaveSeq#:00000001</td>
</tr>
<tr>
<td>+000E00 NextEnvAlri:00000000_00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System Managed Control Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue #: 0000000000000018</td>
</tr>
<tr>
<td>ALESETE: 00000001_00107E18</td>
</tr>
<tr>
<td>+000000 Flags:00000000 ALRI:00000001_0010E0D8</td>
</tr>
<tr>
<td>+000100 WTime:00000000_00000000 InitNum:00000000_0000000A</td>
</tr>
<tr>
<td>+000300 IncNum:00000000_00000005 MaxNum:00000000_00000014</td>
</tr>
<tr>
<td>+000500 CurNum:00000000_0000000A RTOPtr:00000001_00207C0D</td>
</tr>
<tr>
<td>+000700 RTOLen:00000000_00000040</td>
</tr>
</tbody>
</table>

Figure 221: Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 2 of 4)
The following is part three of the example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64).

---

**Routine Control Blocks**

Queue #: 0000000000000017

Routine: ALEM001

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID:ALRI</td>
<td>0000000001_00200140</td>
</tr>
<tr>
<td>RoutineName:</td>
<td>ALEM001</td>
</tr>
<tr>
<td>ALRI:</td>
<td>00000001_00200140</td>
</tr>
<tr>
<td>AleiAddress:</td>
<td>0000000000_00000000</td>
</tr>
<tr>
<td>AleiInstanceNum:</td>
<td>0000000000_00000000</td>
</tr>
<tr>
<td>DllName:</td>
<td>斡</td>
</tr>
<tr>
<td>RoutineNamePtr:</td>
<td>00000001_00200320</td>
</tr>
<tr>
<td>RoutineAddr:</td>
<td>00000000_263840C0</td>
</tr>
<tr>
<td>QSTRTAddr:</td>
<td>00000000_26384000</td>
</tr>
<tr>
<td>DllKeyPtr:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>DllKeyLen:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>ParmLen:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>EnvFlags:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>FuncEnv:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>FuncEntry:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>MasterAlri:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>EnvType:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>ALRI:</td>
<td>00000001_00200340</td>
</tr>
<tr>
<td>RoutineName:</td>
<td>ALEM001</td>
</tr>
<tr>
<td>ALRI:</td>
<td>00000001_00200340</td>
</tr>
<tr>
<td>AleiAddress:</td>
<td>0000000000_00000000</td>
</tr>
<tr>
<td>AleiInstanceNum:</td>
<td>0000000000_00000000</td>
</tr>
<tr>
<td>DllName:</td>
<td>斡</td>
</tr>
<tr>
<td>RoutineNamePtr:</td>
<td>00000001_00200320</td>
</tr>
<tr>
<td>RoutineAddr:</td>
<td>00000000_263840C0</td>
</tr>
<tr>
<td>QSTRTAddr:</td>
<td>00000000_26384000</td>
</tr>
<tr>
<td>DllKeyPtr:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>DllKeyLen:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>ParmLen:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>EnvFlags:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>FuncEnv:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>FuncEntry:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>MasterAlri:</td>
<td>00000000_00000000</td>
</tr>
<tr>
<td>EnvType:</td>
<td>00000000_00000000</td>
</tr>
</tbody>
</table>

---

**Figure 222: Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 3 of 4)**
The following is part four of the example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64).

[10] ALEI: 000000001_0010C0CA8
+000000 ID:ALEI Flags1:00000000 InstanceNum:00000000 00000000
+000010 Next:000000001_0010DA90 Prev:000000000_00000000
+000020 Flags2:00000000 EnclaveSeq#:00000000 LAA:01ED8B18
+00002C SavedLAA:00000000 ALEC:00000000_7F6F7000
+000038 CallerPSWKey:0000000000000000 ASASTack:00000000_00000000
+000048 ParmListPtr:00000000 00000000 ParmListPtrK:00000000 00000000
+000058 RTOPtr:000000001_0010DA90 RTOLEn:00000000 00000401
+000068 RTBL@:00000000 00000000 CallAlri:00000001_00200740
+000078 EnvAlris:00000000 00000000 SystemRtnCode:00000000
+000084 SystemRsnCode:00000000 SystemRtnCodeJr:00000000
+00011C SystemRtnCodeJr:00000000 ALES:00000001_00107CD0
⋮
[8] ETINDEX: 00000002
ALESETE: 000000001_00107FA8
+000000 Flags:00000000 InstanceNum:00000000 00000008
+000010 WTime:00000000 00000000 InitNum:00000000 0000000B
+000020 IncrNum:00000000 00000006 MaxNum:00000000 00000017
+000030 CurNum:00000000 0000000B RTOPtr:00000001_002080D0
+000040 RTOLen:00000000 00000400
[9] Routine Control Blocks
Queue #: 0000000000000017
Routine: ALEM001
ALRI: 000000001_00200540
+000000 ID:ALRI Flags:88000000 InstanceNum:00000000 00000023
+000010 NEXT:000000001_00200740 ALEC:00000000_7F6F7000
+000020 AleiAddress:00000000 00000000 ALEIInstanceNum:00000000 00000000
+000030 OllName:...... RoutineNamePtr:00000001_002080D0
+000040 RoutineNameLen:0000000000000000 RoutineAddr:00000000_263840C0
+000050 QTSTRAddr:00000000 26384000 DllKeyPtr:00000000 00000000
+000060 DllKeyLen:00000000 00000000 ParmLen:0000000000000000
+00006C EnvFlags:00000000 FuncEnv:00000000 00000000
+000078 FuncEntry:00000000 00000000 MasterAlri:00000000 00000000
+00009B ALES:000000001_00107CD0 LUALri:00000001 00200740
+0000FF EnvType:00000002 EnclaveSeq#:00000000
+0010E8 NextEnvAlri:00000000 00000000
Routine: ALEM001
ALRI: 000000001_00200540
+000000 ID:ALRI Flags:88000000 InstanceNum:00000000 00000023
+000010 NEXT:000000001_00200740 ALEC:00000000_7F6F7000
+000020 AleiAddress:00000000 00000000 ALEIInstanceNum:00000000 00000000
+000030 OllName:...... RoutineNamePtr:00000001_002080D0
+000040 RoutineNameLen:0000000000000000 RoutineAddr:00000000_263840C0
+000050 QTSTRAddr:00000000 26384000 DllKeyPtr:00000000 00000000
+000060 DllKeyLen:00000000 00000000 ParmLen:0000000000000000
+00006C EnvFlags:00000000 FuncEnv:00000000 00000000
+000078 FuncEntry:00000000 00000000 MasterAlri:00000000 00000000
+00009B ALES:000000001_00107CD0 LUALri:00000001 00200740
+0000FF EnvType:00000002 EnclaveSeq#:00000000
+0010E8 NextEnvAlri:00000000 00000000
[10] ALEI: 000000001_00112508
+000000 ID:ALEI Flags1:00000000 InstanceNum:00000000 00000015
+000010 Next:000000001_001100D8 Prev:000000000_00000000
+000020 Flags2:00000000 EnclaveSeq#:00000000 LAA:01ED7018
+00002C SavedLAA:7F7018B ALEC:00000000_7F6F7000
+000038 CallerPSWKey:0000000000000000 ASASTack:00000000 7F6C4A0C
+000048 ParmListPtr:00000000 7F7018B ALES:000000001_00107CD0
+000058 RTOPtr:00000001_00200740 RTOLEn:00000000 00000401
+000068 RTBL@:00000000 00000000 CallAlri:00000001 00200740
+000078 EnvAlris:00000000 00000000 SystemRtnCode:00000000
+000084 SystemRsnCode:00000000 SystemRtnCodeJr:00000000
+00011C SystemRtnCodeJr:00000000 ALES:00000001_00107CD0
⋮
[8] ETINDEX: 00000003
ALESETE: 000000001_00108138
+000000 Flags:00000000 InstanceNum:00000000 00000000
+000010 WTime:00000000 00000000 InitNum:00000000 0000001C
+000020 IncrNum:00000000 00000006 MaxNum:00000000 00000021
+000030 CurNum:00000000 0000000B RTOPtr:00000001_00200740
+000040 RTOLen:00000000 00000400
Exiting Language Environment Data

Figure 223: Example of formatted AUTH output from LEDATA VERBEXIT (AMODE 64) (Part 4 of 4)

Sections of the AUTH LEDATA VERBEXIT formatted output

Table 60 on page 416 describes the contents of the AUTH LEDATA VERBEXIT formatted output.
<table>
<thead>
<tr>
<th>Section number and heading</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] ALEC</td>
<td>Anchor control block for all other Preinitialized Environments for Authorized Programs control blocks within the address space. The ALEC is located from the ASXB (Address Space Extension Block).</td>
</tr>
<tr>
<td>[2] Load Module Control Blocks</td>
<td>Formatted representation of a table of ALMI control blocks. Each ALMI represents a module that was loaded by Preinitialized Environments for Authorized Programs.</td>
</tr>
<tr>
<td>[3] User Managed Control Blocks</td>
<td>Control blocks for all user-managed environments. A user-managed environment is initialized when the CELAAUTH macro is invoked with REQUEST=USERINIT.</td>
</tr>
<tr>
<td>[4] ALEI</td>
<td>Each ALEI control block represents one environment. This is a control block for one user-managed environment. This section is repeated for each user-managed environment that was initialized.</td>
</tr>
<tr>
<td>[5] Routine Control Blocks</td>
<td>Formatted representation of a table of ALRI control blocks. Each ALRI in this section represents a routine that was called by the user-managed environment. Each ALRI appears in the table twice, once for the routine name and once for the routine address. This is a control block for one user-managed environment. This section is repeated for each user-managed environment that was initialized.</td>
</tr>
<tr>
<td>[6] System Managed Control Blocks</td>
<td>Control blocks for all system-managed environments. A set of system-managed environments is initialized when the CELAAUTH macro is invoked with REQUEST=MNGDINIT.</td>
</tr>
<tr>
<td>[7] ALES</td>
<td>Each ALES represents a set of system-managed environments. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized.</td>
</tr>
<tr>
<td>[8] ETINDEX and ALESETE</td>
<td>The ETINDEX is the environment definition entry index value and the ALESETE represents the environment definition entry. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized. This is a control block for one environment definition entry. This section is repeated for every environment definition entry (AEDE) that was specified when the set of system-managed environments was initialized.</td>
</tr>
<tr>
<td>[9] Routine Control Blocks</td>
<td>Formatted representation of a table of ALRI control blocks. Each ALRI in this section represents a routine that was called in one of the environments associated with the ETINDEX and ALESETE. Each ALRI appears in the table twice, once for the routine name and once for the routine address. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized. This is a control block for one environment definition entry. This section is repeated for every environment definition entry (AEDE) that was specified when the set of system-managed environments was initialized.</td>
</tr>
<tr>
<td>Section number and heading</td>
<td>Contents</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>[10] ALEI</td>
<td>Each ALEI control block represents one environment. The ALEIs in this section represent system-managed environments. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized. This is a control block for one environment definition entry. This section is repeated for every environment definition entry (AEDE) that was specified when the set of system-managed environments was initialized. This is a control block for one system-managed environment. This section is repeated for every environment that is associated with the ETINDEX and ALESETES.</td>
</tr>
<tr>
<td>[11] ALRI</td>
<td>Contains the ALRI control blocks for each routine that was called in the environment that was identified by the ALEI. This section does not appear if the environment was not used to call a routine. This is a control block for one set of system-managed environments that was initialized. This section is repeated for each set of system-managed environment that was initialized. This is a control block for one environment definition entry. This section is repeated for every environment definition entry (AEDE) that was specified when the set of system-managed environments was initialized. This is a control block for one system-managed environment. This section is repeated for every environment that is associated with the ETINDEX and ALESETES.</td>
</tr>
</tbody>
</table>

### Formatting individual control blocks

In addition to the full LEDATA output which contains many formatted control blocks, the IPCS Control block formatter can also format individual Language Environment control blocks. The IPCS CBF command can be invoked from the "IPCS Subcommand Entry" screen, option 6 of the "IPCS PRIMARY OPTION MENU". For more information on using the IPCS CBF command, see the "CBFORMAT subcommand" section in *z/OS MVS IPCS Commands*.

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

**address**
The address of the control block in the dump. This is determined by browsing the dump or running the LEDATA VERBEXIT.

**cbname**
The name of the control block to be formatted. The control blocks that can be individually formatted are listed in Table 61 on page 418. In general, the name of each control block is similar to that used by the LEDATA VERBEXIT and is generally found in the control block’s eyecatcher field. However, all control block names are prefixed with CEE to uniquely define the Language Environment control block names to IPCS.

For example, the following command produces the output shown in Figure 224 on page 418.

```
CBF 100007B18 struct(CELCAA)
```
Table 61: Language Environment control blocks that can be individually formatted

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELCIB</td>
<td>Condition Information Block</td>
</tr>
<tr>
<td>CELCIBH</td>
<td>Condition Information Block Header</td>
</tr>
<tr>
<td>CELDLLF</td>
<td>DLL Failure Control Block</td>
</tr>
<tr>
<td>CELDSA</td>
<td>Dynamic Storage Area</td>
</tr>
<tr>
<td>CELDSATR</td>
<td>XPLINK Transition Area</td>
</tr>
<tr>
<td>CELEDB</td>
<td>Enclave Data Block</td>
</tr>
<tr>
<td>CELENSQ</td>
<td>Enclave Level Storage Management</td>
</tr>
<tr>
<td>CELHNQ31</td>
<td>Heap Anchor Node 31-bit</td>
</tr>
<tr>
<td>CELHCOM</td>
<td>CEL Exception Manager Communications Area</td>
</tr>
<tr>
<td>CELHPCQ</td>
<td>Thread Level Heap Control Block</td>
</tr>
<tr>
<td>CELLAA</td>
<td>Library Anchor Area</td>
</tr>
<tr>
<td>CELLCA</td>
<td>Library Communication Area</td>
</tr>
<tr>
<td>CELPCB</td>
<td>Process Control Block</td>
</tr>
<tr>
<td>CELRCB</td>
<td>Region Control Block</td>
</tr>
<tr>
<td>CELSANC</td>
<td>Storage Management Control Block</td>
</tr>
<tr>
<td>CELSTS B</td>
<td>Storage Reports Statistics Block</td>
</tr>
</tbody>
</table>

Table 62: Preinitalized Environments for Authorized Programs control blocks that can be individually formatted

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELALEC</td>
<td>Anchor Block</td>
</tr>
<tr>
<td>CELALEI</td>
<td>Environment Information Block</td>
</tr>
</tbody>
</table>
Table 62: Preinitialized Environments for Authorized Programs control blocks that can be individually formatted (continued)

<table>
<thead>
<tr>
<th>Control Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELALES</td>
<td>System Managed Environment Set Block</td>
</tr>
<tr>
<td>CELALMI</td>
<td>Module Information Block</td>
</tr>
<tr>
<td>CELALRI</td>
<td>Routine Information Block</td>
</tr>
</tbody>
</table>

**Requesting a Language Environment trace for debugging**

Language Environment provides an in-storage, wrapping trace facility that can reconstruct the events leading to the point where a dump is taken. Language Environment produces a trace table in its dump report when the TRACE runtime option is set to ON and:

- A thread ends abnormally because of an unhandled condition of severity 2 or greater and the TERMTHDACT runtime option is set to DUMP, UADUMP, TRACE, or UATRACE.
- An application terminates normally and the TRACE runtime option is set to DUMP (the default).

For more information about recording done by the TERMTHDACT runtime option or the TRACE runtime option, see [z/OS Language Environment Programming Reference](#).

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

The contents of the Language Environment dump depend on the values set in the TERMTHDACT runtime option. Table 63 on page 419 summarizes the dump contents that are generated under abnormal termination.

Table 63: TERMTHDACT runtime option settings and dump contents produced (AMODE 64)

<table>
<thead>
<tr>
<th>TERMTHDACT value</th>
<th>Type of dump generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMTHDACT(QUIET)</td>
<td>Language Environment dump containing the trace table only</td>
</tr>
<tr>
<td>TERMTHDACT(MSG)</td>
<td>Language Environment dump containing the trace table only</td>
</tr>
<tr>
<td>TERMTHDACT(TRACE)</td>
<td>Language Environment dump containing the trace table and the traceback</td>
</tr>
<tr>
<td>TERMTHDACT(DUMP)</td>
<td>Language Environment dump containing thread/enclave/process storage and control blocks; the trace table is included as an enclave control block</td>
</tr>
<tr>
<td>TERMTHDACT(UAONLY)</td>
<td>System dump of the user address space and a Language Environment dump that contains the trace table</td>
</tr>
<tr>
<td>TERMTHDACT(UATRACE)</td>
<td>Language Environment dump that contains traceback information, and a system dump of the user address space</td>
</tr>
<tr>
<td>TERMTHDACT(UADUMP)</td>
<td>Language Environment dump containing thread/enclave/process storage and control blocks (the trace table is included as an enclave control block), and a user address space dump</td>
</tr>
<tr>
<td>TERMTHDACT(UAIMM)</td>
<td>System dump of the user address space of the original abend or program interrupt that occurred before the Language Environment condition manager processing the condition. Also contains a Language Environment dump, which contains the trace table. TRAP(ON,NOSPIE) must be in effect. When TRAP(ON,SPIE) is in effect, UAIMM equals UAONLY results. For software raised conditions or signals, UAIMM is the same as UAONLY.</td>
</tr>
</tbody>
</table>
Under normal termination, with the TRACE runtime option set to DUMP, Language Environment generates a dump containing the trace table only, independent of the TERMTHDACT setting.

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

**Locating the trace dump**

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

If your application is running under z/OS UNIX and is either running in a child process, or if it is invoked by one of the exec family of functions, the dump is written to the z/OS UNIX file system. Language Environment writes the CEEDUMP to one of the following directories in the specified order:

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

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

```
/path/CEEDUMP.Date.Time.Pid
```

- **path**
  - The path determined from the above algorithm.

- **Date**
  - The date the dump is taken, appearing in the format YYYYMMDD (such as 20040918 for September 18, 2004).

- **Time**
  - The time the dump is taken, appearing in the format HHMMSS (such as 175501 for 05:55:01 p.m.).

- **Pid**
  - The process ID the application is running in when the dump is taken.

**Using the Language Environment trace table format in a dump report**

The Language Environment trace table is established unconditionally at enclave initialization time if the TRACE runtime option is set to ON. All threads in the enclave share the trace table; there is no thread-specific table, nor can the table be dynamically extended or enlarged.

**Understanding the trace table entry (TTE)**

Each trace table entry is a fixed-length record consisting of a fixed-format portion (containing such items as the timestamp, thread ID, and member ID) and a member-specific portion. The member-specific portion has a fixed length, of which some (or all) can be unused. For information about how participating products use the trace table entry, see the product-specific documentation. The format of the trace table entry is shown in Figure 225 on page 421.
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
Name
01
CEL
03
C/C++
08
Reserved
11
Enterprise PL/I
12
Sockets

Flags
24 flags reserved for internal use.

Member Entry Type
A number that indicates the type of the member-specific trace information that follows the field. To uniquely identify the information contained in a specific TTE, you must consider Member ID as well as Member Entry Type.

Member-Specific Information
Based on the member ID and the member entry type, this field contains the specific information for the entry, up to 104 bytes. For C/C++, the entry type of 1 is a record that records an invocation of a base C runtime library function. The entry consists of the name of the invoking function and the name of the invoked function. Entry type 2 is a record that records the return from the base library function. It contains the returned value and the value of errno.

Member-specific information in the trace table entry
Global tracing is activated by using the LE=n suboption of the TRACE runtime option. This requests all Language Environment members to generate trace records in the trace table. The settings for the global trace events are:

Level
Description
0
No global trace
Trace all runtime library (RTL) function entry and exits

Trace all RTL mutex init/destroy and lock/unlock

Trace all RTL function entry and exits, and all mutex init/destroy and lock/unlock

Trace all RTL storage allocation/deallocation

*When LE=1 is specified*

Table 64 on page 422 shows the C/C++ records that may be generated. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 448.

*Table 64: LE=1 entry records (AMODE 64)*

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

*When LE=2 is specified*

Table 65 on page 422 shows the Language Environment records that may be generated.

*Table 65: LE=2 entry records (AMODE 64)*

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00000101</td>
<td>LT</td>
<td>A</td>
<td>Latch Acquire</td>
</tr>
<tr>
<td>01</td>
<td>00000102</td>
<td>LT</td>
<td>R</td>
<td>Latch Release</td>
</tr>
<tr>
<td>01</td>
<td>00000103</td>
<td>LT</td>
<td>W</td>
<td>Latch Wait</td>
</tr>
<tr>
<td>01</td>
<td>00000104</td>
<td>LT</td>
<td>AW</td>
<td>Latch Acquire after Wait</td>
</tr>
<tr>
<td>01</td>
<td>00000106</td>
<td>LT</td>
<td>I</td>
<td>Latch Increment (Recursive)</td>
</tr>
<tr>
<td>01</td>
<td>00000107</td>
<td>LT</td>
<td>D</td>
<td>Latch Decrement (Recursive)</td>
</tr>
<tr>
<td>01</td>
<td>000002FC</td>
<td>LE</td>
<td>EUO</td>
<td>Latch unowned (not released)</td>
</tr>
<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>
</tbody>
</table>
Table 65: LE=2 entry records (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Class</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0000031D</td>
<td>MX</td>
<td>BI</td>
<td>Shared memory lock init</td>
</tr>
<tr>
<td>01</td>
<td>0000031E</td>
<td>MX</td>
<td>BD</td>
<td>Shared memory lock destroy</td>
</tr>
<tr>
<td>01</td>
<td>0000031F</td>
<td>MX</td>
<td>BO</td>
<td>Shared memory lock obtain</td>
</tr>
<tr>
<td>01</td>
<td>00000320</td>
<td>MX</td>
<td>BC</td>
<td>Shared memory lock obtain on condition</td>
</tr>
<tr>
<td>01</td>
<td>00000321</td>
<td>MX</td>
<td>BR</td>
<td>Shared memory lock release</td>
</tr>
<tr>
<td>01</td>
<td>00000324</td>
<td>MX</td>
<td>CIN</td>
<td>Call to SMC_INIT</td>
</tr>
<tr>
<td>01</td>
<td>00000325</td>
<td>MX</td>
<td>CSD</td>
<td>Call to SMC_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>00000326</td>
<td>MX</td>
<td>CSO</td>
<td>Shared resource obtain</td>
</tr>
<tr>
<td>01</td>
<td>00000327</td>
<td>MX</td>
<td>CSR</td>
<td>Shared resource release</td>
</tr>
<tr>
<td>01</td>
<td>00000328</td>
<td>MX</td>
<td>CST</td>
<td>Call to SMC_SetupToWait</td>
</tr>
<tr>
<td>01</td>
<td>00000329</td>
<td>MX</td>
<td>CSP</td>
<td>Call to SMC_POST</td>
</tr>
<tr>
<td>01</td>
<td>000004CC</td>
<td>ME</td>
<td>FFR</td>
<td>Error - Forced release (shared mutex)</td>
</tr>
<tr>
<td>01</td>
<td>000004CD</td>
<td>ME</td>
<td>FFD</td>
<td>Error - Forced decrement (shared mutex)</td>
</tr>
<tr>
<td>01</td>
<td>000004CE</td>
<td>ME</td>
<td>FBD</td>
<td>Error - BPX_SMC(DESTROY) error return</td>
</tr>
<tr>
<td>01</td>
<td>000004CF</td>
<td>ME</td>
<td>FBU</td>
<td>Error - BPX_SMC(fail) returns EBUSY</td>
</tr>
<tr>
<td>01</td>
<td>000004D0</td>
<td>ME</td>
<td>FIV</td>
<td>Error - BPX_SMC(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000004D4</td>
<td>ME</td>
<td>FDU</td>
<td>Error - Destroy failed (uninitialized) (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000004D5</td>
<td>ME</td>
<td>FP</td>
<td>Error - Program check (shared mutex/CV)</td>
</tr>
<tr>
<td>01</td>
<td>000004DB</td>
<td>ME</td>
<td>ESC</td>
<td>Error - BPX1SMC error return</td>
</tr>
<tr>
<td>01</td>
<td>000004DE</td>
<td>ME</td>
<td>EDL</td>
<td>Shared memory lock returns deadlock</td>
</tr>
<tr>
<td>01</td>
<td>000004DF</td>
<td>ME</td>
<td>EIV</td>
<td>Shared memory lock returns invalid</td>
</tr>
<tr>
<td>01</td>
<td>000004E0</td>
<td>ME</td>
<td>EPM</td>
<td>Shared memory lock returns eperm</td>
</tr>
<tr>
<td>01</td>
<td>000004E1</td>
<td>ME</td>
<td>EAG</td>
<td>Shared memory lock returns again</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>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>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>
<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>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>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>
<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>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>00000719</td>
<td>RW</td>
<td>RB</td>
<td>Read busy (tryread failed)</td>
</tr>
<tr>
<td>01</td>
<td>0000071A</td>
<td>RW</td>
<td>WB</td>
<td>Write busy (trywrite failed)</td>
</tr>
<tr>
<td>01</td>
<td>0000071B</td>
<td>RW</td>
<td>RW</td>
<td>Read wait</td>
</tr>
<tr>
<td>01</td>
<td>0000071C</td>
<td>RW</td>
<td>WW</td>
<td>Write wait</td>
</tr>
<tr>
<td>01</td>
<td>0000071D</td>
<td>RW</td>
<td>BI</td>
<td>Call to SLK_INIT</td>
</tr>
<tr>
<td>01</td>
<td>0000071E</td>
<td>RW</td>
<td>BD</td>
<td>Call to SLK_DESTROY</td>
</tr>
<tr>
<td>01</td>
<td>0000071F</td>
<td>RW</td>
<td>BO</td>
<td>Call to SLK_OBTAIN</td>
</tr>
<tr>
<td>01</td>
<td>00000720</td>
<td>RW</td>
<td>BC</td>
<td>Call to SLK_OBTAIN_COND</td>
</tr>
<tr>
<td>01</td>
<td>00000721</td>
<td>RW</td>
<td>BR</td>
<td>Call to SLK_RELEASE</td>
</tr>
<tr>
<td>01</td>
<td>000008DC</td>
<td>RE</td>
<td>EOW</td>
<td>Error - Already owned for write (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008DD</td>
<td>RE</td>
<td>EOR</td>
<td>Error - Already owned for read (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008DE</td>
<td>RE</td>
<td>EDL</td>
<td>Error - BPX1SLK(fail) returns EDEADLK</td>
</tr>
<tr>
<td>01</td>
<td>000008DF</td>
<td>RE</td>
<td>EIV</td>
<td>Error - BPX1SLK(fail) returns EINVAL</td>
</tr>
<tr>
<td>01</td>
<td>000008E0</td>
<td>RE</td>
<td>EPM</td>
<td>Error - BPX1SLK(fail) returns EPERM</td>
</tr>
<tr>
<td>01</td>
<td>000008E1</td>
<td>RE</td>
<td>EAG</td>
<td>Error - BPX1SLK(fail) returns EAGAIN</td>
</tr>
<tr>
<td>01</td>
<td>000008E2</td>
<td>RE</td>
<td>EBS</td>
<td>Error - BPX1SLK(fail) returns EBUSY</td>
</tr>
<tr>
<td>01</td>
<td>000008E3</td>
<td>RE</td>
<td>ENM</td>
<td>Error - BPX1SLK(fail) returns ENOMEM</td>
</tr>
<tr>
<td>01</td>
<td>000008E4</td>
<td>RE</td>
<td>EBR</td>
<td>Error - BPX1SLK(RELEASE) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E5</td>
<td>RE</td>
<td>EBC</td>
<td>Error - BPX1SLK(OBTAIN_COND) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E6</td>
<td>RE</td>
<td>EBO</td>
<td>Error - BPX1SLK(OBTAIN) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E7</td>
<td>RE</td>
<td>EBD</td>
<td>Error - BPX1SLK(DESTROY) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E8</td>
<td>RE</td>
<td>EBI</td>
<td>Error - BPX1SLK(INIT) error return</td>
</tr>
<tr>
<td>01</td>
<td>000008E9</td>
<td>RE</td>
<td>EFR</td>
<td>Error - Forced release</td>
</tr>
<tr>
<td>01</td>
<td>000008EA</td>
<td>RE</td>
<td>EFD</td>
<td>Error - Forced decrement</td>
</tr>
<tr>
<td>01</td>
<td>000008ED</td>
<td>RE</td>
<td>EIA</td>
<td>Error - Initialization failed (attribute)</td>
</tr>
<tr>
<td>01</td>
<td>000008EE</td>
<td>RE</td>
<td>EIS</td>
<td>Error - Initialization failed (storage)</td>
</tr>
<tr>
<td>01</td>
<td>000008EF</td>
<td>RE</td>
<td>EF</td>
<td>Error - Forced by quiesce</td>
</tr>
<tr>
<td>01</td>
<td>000008F0</td>
<td>RE</td>
<td>EP</td>
<td>Error - Program check</td>
</tr>
<tr>
<td>01</td>
<td>000008FB</td>
<td>RE</td>
<td>EUI</td>
<td>Error - Uninitialized</td>
</tr>
<tr>
<td>01</td>
<td>000008FC</td>
<td>RE</td>
<td>EUO</td>
<td>Error - Unowned (not released)</td>
</tr>
<tr>
<td>01</td>
<td>000008FD</td>
<td>RE</td>
<td>EO</td>
<td>Error - Already owned (not acquired)</td>
</tr>
<tr>
<td>01</td>
<td>000008FE</td>
<td>RE</td>
<td>EIN</td>
<td>Error - Initialization failed (duplicate)</td>
</tr>
</tbody>
</table>

Table 66 on page 427 shows the format for the Mutex – Condition Variable – Latch entries in the trace table.
Table 66: Format of the mutex/CV/latch records (AMODE 64).

<table>
<thead>
<tr>
<th>Record fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
</tr>
<tr>
<td>unused</td>
</tr>
</tbody>
</table>

**Class**
- Two character EBCDIC representation of the trace class.
  - LT
    - Latch
  - LE
    - Latch Exception
  - MX
    - Mutex
  - ME
    - Mutex Exception
  - CV
    - Condition Variable
  - CE
    - Condition Variable Exception

**Source**
- One character EBCDIC representation of the event.
  - C
    - C/C++

**Blank**
- Blank character

**Event**
- Two character EBCDIC representation of the event; see Table 65 on page 422.

**Object addr**
- Fullword address of the mutex object.

**Name 1**
- Optional eight character field containing the name of the function or object to be recorded.

**Name 2**
- Optional eight character field containing the name of the function or object to be recorded.

**When LE=3 is specified**

The trace table will include the records generated by both LE=1 and LE=2.

**When LE=8 is specified**

As Table 67 on page 427 shows, the trace table will contain only storage allocation records. Currently, this is only supported by C/C++. For a detailed description of these records, see “C/C++ contents of the Language Environment trace tables” on page 448.

Table 67: LE=8 entry records (AMODE 64)

<table>
<thead>
<tr>
<th>Member ID</th>
<th>Record Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>00000005</td>
<td>Storage allocation entry</td>
</tr>
<tr>
<td>03</td>
<td>00000006</td>
<td>Storage allocation exit</td>
</tr>
</tbody>
</table>
Sample dump for the trace table entry

Figure 226 on page 428 shows an example of a dump of the trace table when you specify the LE=1 suboption (the library call/return trace).

Figure 226: Trace table in dump output (LE=1 suboption)

Requesting a UNIX System Services syscall trace for debugging

Signal SIGTRACE can be sent to a process or process group to start or stop a trace of the z/OS UNIX System Services syscalls made by the application. The signal is implemented as a toggle. With the trace turned on, the z/OS UNIX System Services kernel gathers the syscall trace records for the targeted processes. A system dump of the user address space can be generated by sending signal SIGDUMP to the same processes in order to capture the trace output. See z/OS UNIX System Services Command Reference for more information about the SIGTRACE signal.
Chapter 13. Debugging AMODE 64 C/C++ routines

This section provides specific information to help you debug AMODE 64 applications that contain one or more C/C++ routines. It includes the following topics:

• Debugging C/C++ I/O routines
• Using XL C/C++ compiler listings
• Generating a Language Environment dump of a C/C++ routine
• Finding C/C++ information in a Language Environment dump
• Debugging example of C/C++ routines

There are several debugging features that are unique to C/C++ routines. Before examining the C/C++ techniques to find errors, you might want to consider the following areas of potential problems:

• To prevent errors that may result from differences in LP64 default argument types, you should include function prototypes for all C/C++ function calls. For C/C++ runtime library functions, see z/OS XL C/C++ Runtime Library Reference.

  Note: malloc() is an example of a RTL function which needs this prototype to work correctly in LP64 applications.

• If you are using the fetch() function, see z/OS XL C/C++ Programming Guide to ensure that you are creating the fetchable module correctly.

• If you are using DLLs, see z/OS XL C/C++ Programming Guide to ensure that you are using the DLL correctly.

• Ensure that the entry point of the load module is CELQSTRT.

• If you suspect that you are using uninitialized storage, you may want to use the STORAGE runtime option.

• You should avoid:
  – Incorrect casting
  – Referencing an array element with a subscript outside the declared bounds
  – Copying a string to a target with a shorter length than the source string
  – Declaring but not initializing a pointer variable, or using a pointer to allocated storage that has already been freed

If a routine exception occurred and you need more information than the condition handler provided, run your routine with the following runtime options, TRAP(ON, NOSPIE) and TERMTHDACT(UAIMM). Setting these runtime options generates a system dump of the user address space of the original abend or program interrupt prior to the Language Environment condition manager processing the condition. After the system dump is taken by the operating system the Language Environment condition manager continues processing.

Debugging C/C++ programs

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

Using the __amrc and __amrc2 structures to debug input/output

__amrc, a structure defined in stdio.h, can help you determine the cause of errors resulting from an I/O operation, because it contains diagnostic information (for example, the return code from a failed VSAM operation). There are two structures:

• __amrc (defined by type __amrc_type
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 227 on page 430 shows the structure as it appears in stdio.h.

```
typedef struct __amrctype {
    union {
        int __error;
        struct {
            unsigned short __syscode,
            __ic;
        } __abend;
        struct {
            unsigned char __fdbk_fill,
            __ic,
            __ftncd,
            __fdbk;
        } __feedback;
        struct {
            unsigned short __svc99_info,
            __svc99_error;
        } __alloc;
        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;
    } __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;
} __amrc_type;
```

Figure 228 on page 430 shows the __amrc2 structure as it appears in stdio.h.

```
struct {
    int __error2;
    char __pad__error2[4];
    FILE *__fileptr;
    int __reserved[6];
} __amrc2_type;
```

Figure 228: __amrc2 structure (AMODE 64)
union { ... } __code
The error or warning value from an I/O operation is in __error, __abend, __feedback, or __alloc. Look at __last_op to determine how to interpret the __code union.

__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 68 on page 432.

__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++ Runtime Library. This field is set when system-level I/O macro processing triggers an ABEND condition. The macro name values (defined in stdio.h) for this field are as follows:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>__AM_BSAM_NOSWITCH</td>
<td>No switch was made.</td>
</tr>
<tr>
<td>__AM_BSAM_UPDATE</td>
<td>The data set is open for update</td>
</tr>
<tr>
<td>__AM_BSAM_BSAMWRITE</td>
<td>The data set is already open for write (or update) in the same C process.</td>
</tr>
<tr>
<td>__AM_BSAM_FBS_APPEND</td>
<td>The data set is recfm=FBS and open for append</td>
</tr>
<tr>
<td>__AM_BSAM_LRECLX</td>
<td>The data set is recfm=LRECLX (used for VBS data sets where records span the largest blocksize allowed on the device)</td>
</tr>
<tr>
<td>Macro</td>
<td>Definition</td>
</tr>
<tr>
<td>-------</td>
<td>------------</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>

[12] __error2
A secondary error code. For example, an unsuccessful rename or remove operation places its reason code here.

[13] __fileptr
A pointer to the file that caused a SIGIOERR to be raised. Use an fldata() call to get the actual name of the file.

[14] __reserved
Reserved for future use.

__last_op values
The __last_op field is the most important of the __amrc fields. It defines the last I/O operation C/C++ was performing at the time of the I/O error. You should note that the structure is neither cleared nor set by non-I/O operations, so querying this field outside of a SIGIOERR handler should only be done immediately after I/O operations. Table 68 on page 432 lists __last_op values you could receive and where to look for further information.

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__IO_INIT</td>
<td>Will never be seen by SIGIOERR exit value given at initialization.</td>
</tr>
<tr>
<td>__BSAM_OPEN</td>
<td>Sets __error with return code from OS OPEN macro.</td>
</tr>
<tr>
<td>__BSAM_CLOSE</td>
<td>Sets __error with return code from OS CLOSE macro.</td>
</tr>
<tr>
<td>__BSAM_READ</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 66) filled in).</td>
</tr>
<tr>
<td>__BSAM_NOTE</td>
<td>NOTE returned 0 unexpectedly, no return code.</td>
</tr>
<tr>
<td>__BSAM_POINT</td>
<td>This will not appear as an error lastop.</td>
</tr>
<tr>
<td>__BSAM_WRITE</td>
<td>No return code (either __abend (errno == 92) or __msg (errno == 65) filled in).</td>
</tr>
<tr>
<td>__BSAM_CLOSE_T</td>
<td>Sets __error with return code from OS CLOSE TYPE=T.</td>
</tr>
<tr>
<td>__BSAM_BLDL</td>
<td>Sets __error with return code from OS BLDL macro.</td>
</tr>
<tr>
<td>__BSAM_STOW</td>
<td>Sets __error with return code from OS STOW macro.</td>
</tr>
<tr>
<td>__TGET_READ</td>
<td>Sets __error with return code from TSO TGET macro.</td>
</tr>
<tr>
<td>__TPUT_WRITE</td>
<td>Sets __error with return code from TSO TPUT macro.</td>
</tr>
<tr>
<td>__IO_DEVTYPE</td>
<td>Sets __error with return code from I/O DEVTYPE macro.</td>
</tr>
<tr>
<td>__IO_RDWRJFCB</td>
<td>Sets __error with return code from I/O RDJFCB macro.</td>
</tr>
<tr>
<td>__IO_TRKCALC</td>
<td>Sets __error with return code from I/O TRKCALC macro.</td>
</tr>
<tr>
<td>__IO_OBTAIN</td>
<td>Sets __error with return code from I/O CAMLST OBTAIN.</td>
</tr>
<tr>
<td>__IO_LOCATE</td>
<td>Sets __error with return code from I/O CAMLST LOCATE.</td>
</tr>
<tr>
<td>__IO_CATALOG</td>
<td>Sets __error with return code from I/O CAMLST CAT. The associated macro is CATALOG.</td>
</tr>
</tbody>
</table>
Table 68: __last_op values and diagnosis information (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__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 when there is not enough room in a record to start any DBCS string or else when a redundant SO is written to the file before an SI. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_SI_TRUNCATE</td>
<td>This occurs only when there was not enough room to start a DBCS string and data was written anyways, with an SI to end it. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_DBCS_UNEVEN</td>
<td>This occurs when an SI is written before the last double byte character is completed, thereby forcing C or C++ to fill in the last byte of the DBCS string with a padding byte X'FE'. Cannot happen if MB_CUR_MAX is 1.</td>
</tr>
<tr>
<td>__C_CANNOT_EXTEND</td>
<td>This occurs when an attempt is made to extend a file that allows writing, but cannot be extended. Typically this is a member of a partitioned data set being opened for update.</td>
</tr>
<tr>
<td>__VSAM_OPEN_FAIL</td>
<td>Set when a low level VSAM OPEN fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_RRDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is RRDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_KSDS</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is KSDS.</td>
</tr>
<tr>
<td>__VSAM_OPEN_ESDS_PATH</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS PATH.</td>
</tr>
<tr>
<td>__VSAM_OPEN_KSDS_PATH</td>
<td>Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is KSDS PATH.</td>
</tr>
<tr>
<td>__VSAM_MODCB</td>
<td>Set when a low level VSAM MODCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_TESTCB</td>
<td>Set when a low level VSAM TESTCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>__VSAM_SHOWCB</td>
<td>Set when a low level VSAM SHOWCB macro fails, sets __rc and __fdbk fields in the __amrc struct.</td>
</tr>
<tr>
<td>Value</td>
<td>Further Information</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__VSAM_GENCB</td>
<td>Set when a low level VSAM GENCB macro fails, sets __rc and __fdbk fields in the</td>
</tr>
<tr>
<td></td>
<td>__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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>__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</td>
</tr>
<tr>
<td></td>
<td>__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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>__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</td>
</tr>
<tr>
<td></td>
<td>(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</td>
</tr>
<tr>
<td></td>
<td>(errno == 65), look at __msg.</td>
</tr>
<tr>
<td>__QSAM_TRUNC</td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td>__QSAM_FREEPOOL</td>
<td>This is an intermediate operation. You will only see this if an I/O abend occurred.</td>
</tr>
<tr>
<td>__QSAM_CLOSE</td>
<td>Sets __error to result of OS CLOSE macro.</td>
</tr>
<tr>
<td>__QSAM_OPEN</td>
<td>Sets __error to result of OS OPEN macro.</td>
</tr>
<tr>
<td>__CMS_OPEN</td>
<td>Sets __error to result of FSOPEN.</td>
</tr>
<tr>
<td>__CMS_CLOSE</td>
<td>Sets __error to result of FSCLOSE.</td>
</tr>
<tr>
<td>__CMS_READ</td>
<td>Sets __error to result of FSREAD.</td>
</tr>
<tr>
<td>__CMS_WRITE</td>
<td>Sets __error to result of FSWRITE.</td>
</tr>
<tr>
<td>__CMS_STATE</td>
<td>Sets __error to result of FSSTATE.</td>
</tr>
<tr>
<td>__CMS_ERASE</td>
<td>Sets __error to result of FSERASE.</td>
</tr>
<tr>
<td>__CMS_RENAME</td>
<td>Sets __error to result of CMS RENAME command.</td>
</tr>
<tr>
<td>__CMS_EXTRACT</td>
<td>Sets __error to result of DMS EXTRACT call.</td>
</tr>
<tr>
<td>__CMS_LINERD</td>
<td>Sets __error to result of LINERD macro.</td>
</tr>
<tr>
<td>__CMS_LINEWRT</td>
<td>Sets __error to result of LINEWRT macro.</td>
</tr>
<tr>
<td>__CMS_QUERY</td>
<td>__error is not set.</td>
</tr>
<tr>
<td>__HSP_CREATE</td>
<td>Indicates last op was a DSPSERV CREATE to create a hiperspace for a hiperspace</td>
</tr>
<tr>
<td></td>
<td>memory file. If CREATE fails, stores abend code in __amrc_code__abend__syscode,</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>memory file during termination. If DELETE fails, stores abend code in <strong>amrc_code</strong></td>
</tr>
<tr>
<td></td>
<td>__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</td>
</tr>
<tr>
<td></td>
<td>code in __amrc_code__abend__syscode, reason code in __amrc_code__abend__rc.</td>
</tr>
</tbody>
</table>
### Table 68: __last_op values and diagnosis information (AMODE 64) (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>__HSP_WRITE</td>
<td>Indicates last op was a HSPSERV WRITE to a hiperspace. If WRITE fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__HSP_EXTEND</td>
<td>Indicates last op was a HSPSERV EXTEND during a write to a hiperspace. If EXTEND fails, stores abend code in __amrc__code__abend__syscode, reason code in __amrc__code__abend__rc.</td>
</tr>
<tr>
<td>__LFS_OPEN</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_CLOSE</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_READ</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_WRITE</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_LSEEK</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
<tr>
<td>__LFS_FSTAT</td>
<td>Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in z/OS UNIX System Services Programming: Assembler Callable Services Reference.</td>
</tr>
</tbody>
</table>

#### Using file I/O tracing to debug C/C++ file I/O problems

You can use file I/O tracing to debug C/C++ file I/O problems. For more information, see Debugging I/O programs 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 229 on page 435 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 229: Example of a routine using perror() (AMODE 64)
Using __errno2() to diagnose application problems

Use the __errno2() function when diagnosing problems in an application program. This function enables z/OS XL C/C++ application programs to access additional diagnostic information, errno2 (errnojr), associated with errno. The __errno2 may be set by the z/OS XL C/C++ runtime library, z/OS UNIX callable services, or other callable services. The errno2 is intended for diagnostic display purposes only and is not a programming interface.

**Note:** Not all functions set errno2 when errno is set. In the cases where errno2 is not set, the __errno2() function may return a residual value. You may use the __err2ad() function to clear errno2 to reduce the possibility of a residual value being returned.

Figure 230 on page 436 is an example of a routine using __errno2() and Figure 231 on page 436 shows the sample output from that routine.

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

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

**Figure 230: Example of a routine using __errno2() (AMODE 64)**

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

**Figure 231: Sample output of routine using __errno2() (AMODE 64)**

Figure 232 on page 436 is an example of a routine using the environment variable _EDC_ADD_ERRNO2_. Figure 233 on page 437 shows the sample output from that routine. For more information about _EDC_ADD_ERRNO2, see z/OS XL C/C++ Programming Guide.

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

int main(void) {
  FILE *fp;

  /* do NOT add errno2 to perror message */
  setenv("_EDC_ADD_ERRNO2", "0", 1);
  fp = fopen("testfile.dat", "r");
  if (fp == NULL) {
    perror("fopen() failed");
    return 0;
  }
}
```

**Figure 232: Example of a routine using _EDC_ADD_ERRNO2 (AMODE 64)**
#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 (f == NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    } return 0;
}

Figure 234: Example of a routine using __err2ad() in combination with __errno2() (AMODE 64)

Figure 235: Sample output of routine using __err2ad() in combination with __errno2() (AMODE 64)

Using C/C++ listings

For a detailed description of available listings, see z/OS XL C/C++ User’s Guide.

Finding variables

You can determine the value of a variable in the routine at the point of interrupt by using the compiled code listing as a guide to its address, then finding this address in the Language Environment dump or system dump. The method you use depends on the storage class of variable.

It is possible for the routine to be interrupted before the value of the variable is placed in the location provided for it. This can explain unexpected values in the dump.
Steps for finding automatic variables

Perform the following steps to find automatic variables in the Language Environment dump or system dump:

1. Determine the name of the automatic variable and the function it is defined in. As an example, we will find the variable aa in the function main from the program cdivzero shown in Figure 69 on page 203.

2. From the compiler listing, locate the variable in the storage offset listing:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Offset</th>
<th>Class</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:

If the base register is R4, the register 4 value is always the DSA address for the function.

If the base register is not R4, the register value must be located from saved registers.

If the Status field indicates Exception, use the saved registers from when the condition occurred. In the Language Environment dump, the saved registers can be found in the Condition information associated with the DSA address in the Condition Information for Active Routines section. In the formatted output from the IPCS VERBEXIT LEDATA CM subcommand for a system dump, the saved registers can be found in the CIBH that has the DSA address as the value for the SV1 field.

If the Status field indicates Call, use the saved registers from the DSA address that appears on the line above the function in the Traceback. In the Language Environment dump, the DSAs can be found in the "Control Blocks for Active Routines" section. In the formatted output from the IPCS VERBEXIT LEDATA STACK subcommand for a system dump, the DSAs can be found in the "DSA backchain" section.

Note: Some functions do not save all registers.

4. Add the register value to the offset of the variable to obtain the address of the variable. In the Language Environment dump, the contents of the variable can be read in the DSA Frame section corresponding to the function the variable is contained in. For a system dump, use the IPCS LIST subcommand to display the storage where the variable is located.

   The address for variable aa is X'1082FF080' + X'980' = X'1082FFA00'.

Restriction: The parameter value might never be stored, since the first few parameters might be passed in registers and there might be no need to save them.

Steps for finding C/C++ parameters

The C/C++ parameter list is always located in the caller's DSA at offset 2176 (X'880'). Parameters that are passed in registers are not always stored in the parameter list. The compiler option XPLINK(STOREARGS) can be used to ensure that all parameters are stored in the parameter list.

Perform the following steps to find parameters in the Language Environment dump or system dump:

1. Determine the name of the parameter and the function it is for. As an example, we will find the parameter pp for the function funcb from the program cdivzero shown in Figure 53. C routine with a divide-by-zero error.
2. From the compiler listing, locate the parameter in the storage offset listing:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Class</th>
<th>Location</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>pp</td>
<td>parameter</td>
<td>2432(r4)</td>
<td>8</td>
</tr>
</tbody>
</table>

3. From the Traceback (in the Language Environment dump or in the formatted output from the IPCS VERBEXIT LEDATA 'CEEDUMP' subcommand for a system dump) locate the function:

<table>
<thead>
<tr>
<th>Entry</th>
<th>E_Offset</th>
<th>Load Mod</th>
<th>Program Unit</th>
<th>Service</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000003</td>
<td>000002E</td>
<td>CDIVZERO</td>
<td></td>
<td></td>
<td>Exception</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 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 236 on page 439 shows an example of an aggregate.

```c
typedef struct {
    int asid;
    void *addr;
    asfAmodeType amode;
} asfTargetRef;
```

*Figure 236: Example code for structure variables (AMODE 64)*

Figure 237 on page 440 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:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Storage Offset</th>
<th>Class</th>
<th>Location</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>tempTargetRef</td>
<td>209-0:209</td>
<td>automatic</td>
<td>2264(r4)</td>
<td>24</td>
</tr>
</tbody>
</table>

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'1082FD3E0' + X'8D8' = X'1082FDCC0'. 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 Figure 237 on page 440. The offset is 8. Add the offset from the Aggregate Map to the address of the `tempTargetRef` variable.

The result is X'1082FDCC0' (X'1082FD3E0' + 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 an IEATDUMP dump, a Language Environment formatted dump is also taken.
**csnap()**

The `csnap()` function produces a condensed storage dump. To use these functions, you must add `#include <ctest.h>` to your C/C++ code. The dump is directed to output `dumpname`, which is specified in a `//CEEDUMP DD` statement in JCL.

See the [z/OS XL C/C++ Runtime Library Reference](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.7/com.ibm.zos.v1r11.ceq.pdf) 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 238 on page 442 shows a sample C routine that uses the `cdump` function to generate a dump. Figure 244 on page 445 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\n");
    fprintf(fp2, "some more data\n");
    fprintf(fp2, "even more data\n");

    signal(SIGFPE, hsigfpe);
    signal(SIGTERM, hsigterm);

    rc = atexit(atf1);
    if (rc) {
        fprintf(stderr, "Failed on registration of atexit function atf1\n");
        exit(103);
    }
    fetchPtr = (FuncPtr_T) fetch("MODULE1");
    if (!fetchPtr) {
        fprintf(stderr, "Failed to fetch MODULE1\n");
        exit(104);
    }
    fetchPtr();
    return(0);
}
```

Figure 238: Example C routine using cdump() to generate a dump (AMODE 64) (Part 1 of 2)

```c
void hsigfpe(int sig) {
    ++st1;
    return;
}

void hsigterm(int sig) {
    ++st2;
    return;
}

void atf1() {
    ++xcount;
}
```

Figure 239: Example C routine using cdump() to generate a dump (AMODE 64) (Part 2 of 2)

Figure 240 on page 443 shows a fetched C module.
Sample C++ routine that generates a Language Environment dump

Figure 241 on page 443 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 241: Example C++ routine with protection exception generating a dump (AMODE 64)

Figure 242 on page 443 shows the template file stack.c

```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 242: Template file STACK.C (AMODE 64)

Figure 243 on page 444 shows the header file stack.h.
Sample Language Environment dump with C/C++-specific information

This sample dump was produced by compiling the routines shown in Figure 238 on page 442 and Figure 240 on page 443. 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 240 on page 443. func1 calls the library routine __cdump() in statement number 5. The complete program unit names for main and func1 are shown in the Fully Qualified Names section along with its load module name.
Figure 244: Example dump from sample C routine (AMODE 64) (Part 1 of 4)
The following is part two of the example dump from sample C routine (AMODE 64).

Figure 245: Example dump from sample C routine (AMODE 64) (Part 2 of 4)
Figure 246: Example dump from sample C routine (AMODE 64) (Part 3 of 4)
The following is part four of the example dump from sample C routine (AMODE 64).

C/C++ contents of the Language Environment trace tables

Language Environment provides C/C++ trace table entry types 5 and 6, which contain character data.
Trace entry 5 occurs when a C library function is called. The format for trace table entry 5 is:

```
  NameOfCallingFunction
  --> (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.

Trace entry 6 occurs when a C library function returns. The format for trace table entry 6 is:

```
<-- (xxxx) R1=xxxxxxxxxxxxxxxx R2=xxxxxxxxxxxxxxxx R3=xxxxxxxxxxxxxxxx
ERRNO=xxxxxxxx ERRNO2=xxxxxxxx
```

In the entry types, (xxx) and (xxxx) are numbers associated with the called library function and are used to associate a specific entry record with its corresponding return record.

For entry types 5 and 6, the number will be the same as the number of the function as seen in the C runtime library definition side-deck, SCEELIB dataset member CELQS003, on the IMPORT statement for that function.

Figure 248 on page 450 shows an XPLINK trace that contains examples of the trace entries 5 and 6. For more information about the Language Environment trace table format, see “Understanding the trace table entry (TTE)” on page 420.
Figure 248: Trace table with XPLINK trace table entries 5 and 6 (AMODE 64)
The following is the second part of the trace table with XPLINK trace table entries 5 and 6 (AMODE 64).

<table>
<thead>
<tr>
<th>Time 21.58.20.255245</th>
<th>Date 2004.04.20</th>
<th>Thread ID... 8000000000000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>+000480</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000490</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000498</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000488</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3=0000000100000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0004D8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0004F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000510</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000518</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000538</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000558</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2=000000010871AF80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000578</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000590</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000598</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000600</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3=0000000100000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000610</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000618</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000638</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000658</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2=000000010830EA50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000678</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000690</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+000698</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0006B8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3=0000000100000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0006D8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0006F8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 249: Trace table with XPLINK trace table entries 5 and 6 (AMODE 64)

Debugging examples of C/C++ routines

This section contains examples that demonstrate the debugging process for C/C++ routines. Important areas of the output are highlighted. Data unnecessary to the debugging examples has been replaced by ellipses.

Divide-by-zero error

Figure 250 on page 452 illustrates a C program that contains a divide-by-zero error. The code was compiled with RENT so static and external variables need to be calculated from the WSA field. The code was compiled with LP64, GONUM (to produce statement numbers) and XREF, LIST and OFFSET to generate a listing, which is used to calculate addresses of functions and data. The code was processed by the binder with MAP to generate a binder map, which is used to calculate the addresses of static and external variables. The program was created with the option TERMTHDACT(UADUMP) which produced both a Language environment dump and a system dump.
/* C Routine with a Divide-by-Zero Error */
#pragma options(noinline)
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int statint = 73;
int fa;
int funcb(int *pp);
int main(void) {
    int aa, bb=1;
    aa = bb;
    aa = funcb(&aa);
    return(aa);
}
int funcb(int *pp) {
    int result;
    fa = *pp;
    result = fa/(statint-73);
    printf("Result = %d\n",result);
    return result;
}

Figure 250: C routine with a divide-by-zero error (AMODE 64)

To debug this routine, use the following steps:

1. Locate the Original Condition message in the Condition Information for Active Routines section of the dump. In this example, the message is CEE3209S. The system detected a fixed-point divide exception. This message indicates the error was caused by an attempt to divide by zero. For additional information about CEE3209S, see z/OS Language Environment Runtime Messages.

   The traceback section of the dump indicates that the exception occurred at offset X'52' within function funcb. This information is used along with the compiler-generated Pseudo Assembly Listing to determine where the problem occurred.

   If the GONUMBER compiler option is specified, statement number information is in the dump. Figure 251 on page 453 shows the generated traceback from the dump.
2. In the traceback, statement number 12, corresponding to DSA 7, refers to line: `aa = funcb(&aa);` in the listing. This is where entry `funcb` is called. Similarly, statement number 18, corresponding to DSA 6, points to line: `result = fa/(statint-73);` in the listing. This line is where the divide by zero exception takes place.

3. Locate the instruction with the divide-by-zero error in the Pseudo Assembly Listing in Figure 253 on page 454. The offset (within `funcb`) of the exception from the traceback (X'52') reveals the divide instruction: DR R6, R0 at that location. Instructions at offsets X'32' through X'58' refer to the `result = fa/(statint-73);` line of the C/C++ routine.

Figure 251: Sections of the dump from example C/C++ routine (AMODE 64) (Part 1 of 2)

Figure 252: Sections of the dump from example C/C++ routine (AMODE 64) (Part 2 of 2)
OFFSET OBJECT CODE  LINE#  P S E U D O   A S S E M B L Y   L I S T I N G
000010 F2F0  F0F6  =C'2006'  Compiled Year
000014 F0F2  F2F1  =C'0221'  Compiled Date MMDD
000018 F1F5  F3F3  F1F5  =C'153315'  Compiled Time HHHMMSS
00001E F0F1  F0F8  F0F0  =C'010800'  Compiler Version

Timestamp and Version End

000028 00C300C5 =F'12779717'  XPLink entrypoint marker
000030 00000108 =F'264'
000034 00000100 =F'356'
000000 EB59 4708 0024  STMG r5,r9,1800(r4)
000006 A748 FF00 000015  AGHI x4,H'-256'

End of Prolog

000000 C990 0000 060F 000000  LARL x9,F'111'
00000D E310 4980 0024 000015  * int funcb(int *pp) {
00000A EB59 4708 0004 000017  STMG r5,r9,1800(r4)
000010 E300 6000 0014 000017  LG r0,(*)int(,r6,0)
000022 E360 4808 0004 000017  LG r6,#Save_ADA_Ptr_2(,r4,2056)
000028 E360 6600 0004 000017  LG r6,=A(fa)(,r6,0)
000030 E360 6000 0004 000018  STG r1,pp(,r4,2432)
000040 E360 6000 0014 000018  LG r6,=A(statint)(,r7,8)
000044 E300 7000 0014 000018  LGF r0,statint(,r7,0)
00004A A70B FF07 000018  AGHI x0,H'-73'
00004E BE66 0020 00018  SRDA r6,32
000052 1D60 00018  DR r6,0
000054 B964 0007 00018  LGF r0,r7
000058 5000 0008 00018  STG r7,(r6,0)
00005C EB55 4708 0004 00019  STG r5,r9,1800(r4)
000060 E360 4808 0004 00019  LG r6,#Save_ADA_Ptr_2(,r4,2056)
000064 E300 7000 0014 00019  LG r0,statint(,r7,0)
00006A E300 7000 0014 00020  LGF r0,(*)int(,r7,0)
000070 E370 4808 0004 00020  LG r7,=A(fa)(,r6,0)
00007C E370 7900 0004 00020  LG r7,=A(statint)(,r7,8)
000082 E300 7000 0004 00020  LGF r0,statint(,r7,0)
000088 E300 7000 0004 00020  LGF r0,(*)int(,r7,0)
00008E EB82 0024 00018  LRFA r6,0
000090 0D76 00018  BASR r7,r6
000092 0700 00019  NOPR 0
000096 E330 4808 0014 00020  LGF r3,(r4,2240)
00009C 0700 00019  NOPR 0

Start of Epilog

0000A0 000015  @L13  DS  00
0000A0 000015  @L13  DS  00
0000A0 000015  @L13  DS  00

Figure 253: Pseudo assembly listing (AMODE 64) (Part 1 of 3)
The following is part two of the pseudo assembly listing (AMODE 64).

```c
00001 | /* C Routine with a Divide-by-Zero Error from LE Debugging Guide */
00002 | #pragma options(noinline)
00003 | #include <stdio.h>
00004 | #include <stdlib.h>
00005 | #include <errno.h>
00006 | int statint = 73;
00007 | int fa;
00008 | int funcb(int *pp);
00009 | int main(void) {

00010 |     int aa, bb=1;
00011 |     aa = bb;
00012 |     aa = funcb(4);
00013 |     return(aa);

00014 | }
```

Figure 254: Pseudo assembly listing (AMODE 64) (Part 2 of 3)
The following is part three of the pseudo assembly listing (AMODE 64).

PPA1: Entry Point Constants

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>02</td>
</tr>
<tr>
<td>000001</td>
<td>CE</td>
</tr>
<tr>
<td>000002</td>
<td>0700</td>
</tr>
<tr>
<td>000003</td>
<td>00000068</td>
</tr>
<tr>
<td>000004</td>
<td>=AL1(2) Version</td>
</tr>
<tr>
<td>000005</td>
<td>=AL1(206) CEL signature</td>
</tr>
<tr>
<td>000006</td>
<td>=H'1792' GPR save mask</td>
</tr>
<tr>
<td>000007</td>
<td>=F'-21390443999' Flags</td>
</tr>
<tr>
<td>000008</td>
<td>=H'0' Pass length/4</td>
</tr>
<tr>
<td>000009</td>
<td>=H'1283' Prol len/2; alloc reg; R4 change offset/2</td>
</tr>
<tr>
<td>00000A</td>
<td>=F'92' Code length</td>
</tr>
<tr>
<td>00000B</td>
<td>=F'16777216' Interface mapping flags</td>
</tr>
<tr>
<td>00000C</td>
<td>=F'-2139094399' Flags</td>
</tr>
<tr>
<td>00000D</td>
<td>=A(PPA2-PPA1)</td>
</tr>
<tr>
<td>00000E</td>
<td>=A(PPA2-PPA1)</td>
</tr>
<tr>
<td>00000F</td>
<td>=A(PPA2-PPA1)</td>
</tr>
</tbody>
</table>

PPA1 End

PPA4: Compile Unit Debug Block

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>80000000</td>
</tr>
<tr>
<td>000004</td>
<td>=F'-2147483648' Additional Flags</td>
</tr>
<tr>
<td>000008</td>
<td>=F'-2079981960' Flags</td>
</tr>
<tr>
<td>000010</td>
<td>=D'0' R/O static Offset</td>
</tr>
<tr>
<td>000012</td>
<td>=D'0' R/W static Offset</td>
</tr>
<tr>
<td>000018</td>
<td>=D'0' Symbol Offset Table Offset</td>
</tr>
<tr>
<td>000020</td>
<td>=D'-384' CSECT Start Offset</td>
</tr>
<tr>
<td>000022</td>
<td>=D'0' Code CSECT Size</td>
</tr>
<tr>
<td>000030</td>
<td>=D'-376' No program region</td>
</tr>
<tr>
<td>000038</td>
<td>=F'0' DWARF File Name</td>
</tr>
<tr>
<td>00003C</td>
<td>C''</td>
</tr>
</tbody>
</table>

PPA4 End

PPA2: Compile Unit Block

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0300 2204</td>
</tr>
<tr>
<td>000004</td>
<td>FFFF FE40</td>
</tr>
<tr>
<td>000008</td>
<td>FFFF FFC0</td>
</tr>
<tr>
<td>00000C</td>
<td>FFFF FE50</td>
</tr>
<tr>
<td>000010</td>
<td>0000 0000</td>
</tr>
<tr>
<td>000014</td>
<td>9140 0000</td>
</tr>
</tbody>
</table>

PPA2 End

Figure 255: Pseudo assembly listing (AMODE 64) (Part 3 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 256 on page 456 shows the WSA address.

Enclave Control Blocks:

<table>
<thead>
<tr>
<th>Address</th>
<th>Module Addr</th>
<th>Thread ID</th>
<th>Use Count</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>00000001</td>
<td>main</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 256: C/C++ CAA information in dump (AMODE 64)
5. Routines compiled with the RENT option must also be processed by the binder. The binder produces the Writable Static Map. Find the offset of statint in the Writable Static Map in Figure 257 on page 457. In this example, the offset is X'30'.

---

CLASS C_WSA64 LENGTH = 38 ATTRIBUTES = MRG, DEFER, RMODE= 64
OFFSET = 0 IN SEGMENT 002 ALIGN = QQWORD

<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>0</td>
<td>PRIV00012</td>
<td>PART</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>CDIVZERO#C</td>
<td>PART</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>statint</td>
<td>PART</td>
<td>4</td>
<td>statint</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>fa</td>
<td>PART</td>
<td>4</td>
<td>fa</td>
<td></td>
</tr>
</tbody>
</table>

Figure 257: Writable static map (AMODE 64)

---

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 258: IPCS storage display of the writeable static area (AMODE 64)

---

Calling a nonexistent function

Figure 259 on page 458 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).
/* C/C++ Example of Calling a Nonexistent Subroutine */
/* from LE Debugging Guide */
#pragma options(noinline)
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
void funca(int* aa);
int (*func_ptr)(void)=0;

int main(void) {
    int aa;
    funca(&aa);
    printf("result of funca = %d\n",aa);
    return;
}

void funca(int* aa) {
    *aa = func_ptr();
    return;
}

Figure 259: C/C++ example of calling a nonexistent subroutine (AMODE 64)

To debug this routine, use the following steps:

1. Locate the Original Condition message in the Condition Information for Active Routines section of the
dump, which is shown in Figure 260 on page 459. In this example, the message is CEE3201S The
system detected an operation exception (System Completion Code=0C1). This
message suggests that the error was caused by an attempt to branch to an unknown address. For
additional information about CEE3201S, see z/OS Language Environment Runtime Messages.

The Location section of the dump indicates that the exception occurred at offset X'-209000D0' within
function funca and that there might 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 that is branched outside the bounds of the routine.

In the traceback, the statement number that is displayed for entry 'main' points to line 12 in the source
code that is shown in Figure 259 on page 458. This line contains the statement "funca(&aa); " in
which entry 'funca' is called. As message CEE3841I explains, for entry funca no statement number
could be displayed. In this example, this problem is caused because funca has an invalid offset. For
further information about this message see z/OS Language Environment Runtime Messages.
Information for enclave main

Information for thread 0000000000000000

Exception Information

Enclave Control Blocks:

Storage around GPR7 (0000000020900108)

+0030 0000000020900138  EB494700 0024A74B FF000D80 C0900000  |......x.........|
+0020 0000000020900128  00C300C5 00C500F1 000000B0 00000100  |.C.E.E.1........|
+0010 0000000020900118  47F08040 EB484800 000447F0 70020000  |.0. .......0....|

Storage around GPR5 (000A0000000130E1)

+0030 000A000000013111  Inaccessible storage.
+0020 000A000000013101  Inaccessible storage.
+0010 000A0000000130D1  Inaccessible storage.
+0000 000A0000000130E1  Inaccessible storage.
+0000 000A0000000130F1  Inaccessible storage.

Storage around GPR3 (0000000108400070)

+0030 0000000108401F90  C3C4D3C7 6DC8C4D9 00000001 08401FBC  |CDLG_HDR..... ..|
+0020 0000000108401F80  00000001 08400000 00000000 000014E0  |..... ..........|
+0010 0000000108401F70  00000000 00000000 00000000 00000000  |................|
+0000 0000000108401F60  00000001 08401BF0 00000000 00000000  |..... .0........|

Condition Information for Active Routines

The following is part two of sections of the dump from example C routine (AMODE 64).

Storage around GPR2 (0000000108401F60)

-0020 0000000108400050  00000000 00000000 00000000 00000040  |................|

Storage around GPR1 (0000000108401F50)

-0010 0000000108400060  00000001 08400000 00000000 000000C0  |..... ..........|

Figure 260: Sections of the dump from example C routine (AMODE 64) (Part 1 of 3)

Figure 261: Sections of the dump from example C routine (AMODE 64) (Part 2 of 3)
The following is part three of sections of the dump from example C routine (AMODE 64).

<table>
<thead>
<tr>
<th>Process Control Blocks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCB(0000000100003CA0)</td>
</tr>
<tr>
<td>+0000 0000000100003CA0 C3C5C5D7 C3C24040 00000000 00000000</td>
</tr>
</tbody>
</table>
| +0010 0000000100003CB0 00000000 00000000 00000000 00000000 | ... ... ... ... ...
| +0020 0000000100003CC0 +0000FF 0000000100003D9F             | same as above    |
| +0100 0000000100003DA0 03030208 00000000 00000000 00000000 | ... ... ... ... ...
| +0110 0000000100003DB0 00000001 00004048 00000000 00000000 | ...... ......... |
| +0120 0000000100003DC0 00000000 00000000 00000000 00000000 | ... ... ... ... ...
| +0130 0000000100003DD0 00000000 00000000 00000001 00003A10 | ... ... ... ... ...
| +0140 0000000100003DE0 7F800000 00000000 00000000 00000000 | "............... |
| +0150 0000000100003DF0 00000000 00000000 00000000 00000000 | ... ... ... ... ...
| +0160 0000000100003E00 +0001BF 0000000100003E5F             | same as above    |

<table>
<thead>
<tr>
<th>MEML(0000000100004048)</th>
</tr>
</thead>
</table>
| +0000 0000000100004048 00000000 00000000 00000000 00000000 | ... ... ... ... ...
| +0010 0000000100004058 +00005F 00000001000040A7             | same as above    |
| +0060 00000001000040A8 00000001 00008688 00000000 00000000 | ......fh........ |
| +0070 00000001000040B8 00000000 00000000 00000000 00000000 | ... ... ... ... ...
| +0080 00000001000040C8 +0001AF 00000001000041F7             | same as above    |

CEE3846I CEEDUMP Processing completed.

Figure 262: Sections of the dump from example C routine (AMODE 64) (Part 3 of 3)
Figure 263: Pseudo assembly listing (AMODE 64) (Part 1 of 2)

Figure 264: Pseudo assembly listing (AMODE 64) (Part 2 of 2)
3. Find the offset of func_ptr in the Writable Static Map, shown in Figure 265 on page 462.

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_WSA64</th>
<th>LENGTH = 48</th>
<th>ATTRIBUTES = MRG, DEFER, RMODE=</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET= 0</td>
<td>IN SEGMENT 002</td>
<td>ALIGN =</td>
<td></td>
</tr>
</tbody>
</table>

Figure 265: Writable static map (AMODE 64)

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 263 on page 461). 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 266 on page 462 shows the sections of the dump.

<table>
<thead>
<tr>
<th>LIST</th>
<th>01_08300050. ASID(X'00CC') LENGTH(X'0100') AREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>.3000050. C360E6E2 C1F6F440 40404040 40404040</td>
<td>C_WSA64</td>
</tr>
<tr>
<td>.3000060. 94819995 0064A995 08010000 00000000</td>
<td>main_funca...</td>
</tr>
<tr>
<td>.3000070. 00000000 08300000 00000000 00000000</td>
<td>...</td>
</tr>
<tr>
<td>.3000080. 00000000 00000000 00000000 20E71FF8</td>
<td>... (X.8)</td>
</tr>
<tr>
<td>.3000090 LENGTH(X'10')=&gt;$All bytes contain X'00'</td>
<td></td>
</tr>
<tr>
<td>.30000A0. 00000000 08300000 00000000 00000002</td>
<td></td>
</tr>
<tr>
<td>.30000B0. 00000000 08300200 00000001 08300488</td>
<td></td>
</tr>
<tr>
<td>.30000C0. 00000000 083004F5 00000001 08300533</td>
<td></td>
</tr>
<tr>
<td>.30000D0. 00000000 0830056F 00000001 083005AC</td>
<td></td>
</tr>
<tr>
<td>.30000E0. 00000000 0830056F 00000001 08300626</td>
<td></td>
</tr>
<tr>
<td>.30000F0. 00000000 08300663 00000001 08300AF0</td>
<td></td>
</tr>
<tr>
<td>.3000100. 00000000 08300AA0 00000000 00000000</td>
<td></td>
</tr>
<tr>
<td>.3000110 LENGTH(X'40')=&gt;$All bytes contain X'00'</td>
<td></td>
</tr>
</tbody>
</table>

Figure 266: IPCS storage display of the writeable static area (AMODE 64)

Handling dumps written to the z/OS UNIX file system

When a z/OS UNIX C/C++ application program is running in an address space created as a result of a call to spawn(), vfork(), or one of the exec family of functions, the SYSMDUMP DD allocation information is not inherited. Even though the SYMDUMP allocation is not inherited, a SYMDUMP allocation must exist in the parent in order to obtain a HFS storage dump. If the program terminates abnormally while running in this new address space, the kernel causes an unformatted storage dump to be written to an HFS file in the user's working directory. The file is placed in the current working directory or into /tmp if the current working directory is not defined. The file name has the following format:

/directory/coredump.pid

where directory is the current working directory or tmp, and pid is the hexadecimal process ID (PID) for the process that terminated. For details on how to generate the system dump, see “Steps for generating a system dump in a z/OS UNIX shell” on page 355.

To debug the dump, use the Interactive Problem Control System (IPCS). If the dump was written to an HFS file, you must allocate a data set that is large enough and has the correct attributes for receiving a copy of the HFS file. For example, from the ISPF DATA SET UTILITY panel you can specify a volume serial
and data set name to allocate. Doing so brings up the DATA SET INFORMATION panel for specifying characteristics of the data set to be allocated.

Figure 267 on page 463 is a sample filled-in panel that shows the characteristics defined for the URCOMP.JRUSL.COREDUMP dump data set. Fill in the information for your data set as shown, and estimate the number of cylinders required for the dump file you are going to copy.

```
-------------------------- DATA SET INFORMATION ----------------------
Command ==>  
Data Set Name . . . : URCOMP.JRUSL.COREDUMP  
General Data Current Allocation
  Management class . . : STANDARD       Allocated cylinders : 30
  Storage class . . . : OS390          Allocated extents . : 1
  Volume serial . . . : DPXDU1
  Device type . . . : 3380
  Data class . . . :  
  Organization . . : P5                     Current Utilization
  Record format . . : FB                     Used cylinders . . : 0
  Record length . . : 4160                   Used extents . . . : 0
  Block size . . . : 4160
  1st extent cylinders: 30  
  Secondary cylinders : 10
  Data set name type :  
  Creation date . . : 2001/08/30  
  Expiration date . . : ***None***
F1=Help     F2=Split     F3=End       F4=Return    F5=Rfind     F6=Rchange
F7=Up       F8=Down      F9=Swap     F10=Left     F11=Right    F12=Cancel
```

Figure 267: IPCS panel for entering data set information (AMODE 64)

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 356 for information about formatting Language Environment control blocks.

**Multithreading consideration**

Certain control blocks are locked while a dump is in progress. For example, a csnap() of the file control block would prevent another thread from using or dumping the same information. An attempt to do so causes the second thread to wait until the first one completes before it can continue.

**Understanding C/C++ heap information in storage reports**

Storage reports that contain specific C/C++ heap information can be generated in two ways; details on how to request and interpret the reports are provided in the following sections.

- By setting the Language Environment RPTSTG(ON) runtime option for Language Environment created heaps
- By issuing a stand-alone call to the C function __uheapreport() for user-created heaps.

**Language Environment storage report with heap pools statistics**

To request a Language Environment storage report set RPTSTG(ON). If the C/C++ application specified the HEAPPOLLS(ON) or HEAPPOLLS64(ON) runtime option, the storage report displays heap pools statistics. For a sample storage report showing heap pools statistics for a multithreaded C/C++ application, see Figure 157 on page 326. The following sections describe the C/C++ specific heap pools information.
HEAPPOOLS64 storage statistics

The HEAPPOOLS64 runtime option controls usage of the heap pools storage algorithm at the enclave level. The heap pools algorithm allows for the definition of one to twelve heap pools, each consisting of a number of storage cells of a specified length.

Note: The use of an alternative vendor heap manager (VHM) overrides the use of the HEAPPOOLS64 runtime option.

HEAPPOOLS64 statistics

- Pool $p$ size: $ssss$ Get requests: $gggg$
  
  $p$
  number of the pool. When there are multiple pools for a cell size, the pools are numbered using the format $aa.bbb$
  
  $aa$
  number for the cell size
  
  $bbb$
  number for the pool within the cell size

  $ssss$
  cell size specified for the pool

  $gggg$
  number of storage requests that were satisfied from this pool

- Successful Get Heap requests: $xxxx-yyyy n$
  
  $xxxx$
  low side of the 8 byte range

  $yyyy$
  high side of the 8 byte range

  $n$
  number of requests in the 8 byte range

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

Note: Values displayed in the HEAPPOOLS64 statistics report are not serialized when collected; therefore, the values are not necessarily exact.

HEAPPOOLS64 summary

The HEAPPOOLS64 summary displays a report of the HEAPPOOLS64 statistics and provides suggested cell sizes.

Specified Cell Size
  the size of the cell specified in the HEAPPOOLS64 runtime option

Element Size
  the size of the cell plus any additional storage needed for control information or to maintain alignment

Cells Per Extent
  the cell pool count specified by the HEAPPOOLS64 runtime option. When there is more than one pool for a cell size, the count is divided by the number of pools.

Extents Allocated
  the number of times that each pool allocated an extent in order to optimize storage usage. The extents allocated needs to be either one or two. If the number of extents allocated is too high, increase the cell count for the pool.

Maximum Cells Used
  the maximum number of cells used for each pool.

Cells In Use
  the number of cells that were never freed. A large number in this field could indicate a storage leak.
Suggested Cell Sizes
sizes that are calculated to optimally use storage (assuming that the application will __malloc/__free
with the same frequency). The suggested cell sizes are given with no cell counts because the usage of
each new cell pool size is not known. If there are less than 12 cell sizes calculated, then the last pool
size is set at 65536.

For more information about stack and heap storage for AMODE64 applications, see z/OS Language

HEAPPOOLS storage statistics

The HEAPPOOLS runtime option controls usage of the heap pools storage algorithm at the enclave level.
The heap pools algorithm allows for the definition of one to twelve heap pools, each consisting of a
number of storage cells of a specified length. HEAPPOOLS runtime option can be used by AMODE 64
applications to manage user heap storage above the 16MB line and below the 2GB bar.

Note: The use of an alternative Vendor Heap Manager (VHM) overrides the use of the HEAPPOOLS
runtime option.

HEAPPOOLS statistics

• Pool p size: ssss Get requests: gggg
  p
  number of the pool. When there are multiple pools for a cell size, the pools are numbered using the
  format aa.bbb
  aa
  number for the cell size
  bbb
  number for the pool within the cell size
  ssss
  cell size specified for the pool
  gggg
  number of storage requests that were satisfied from this pool
• Successful Get Heap requests: xxxx-yyyy n
  xxxx
  low side of the 8 byte range
  yyyy
  high side of the 8 byte range
  n
  number of requests in the 8 byte range
• Requests greater than the largest cell size — the number of storage requests that are not
  satisfied by heap pools.

Note: Values displayed in the HEAPPOOLS statistics report are not serialized when collected, therefore
the values are not necessarily exact.

HEAPPOOLS summary

The HEAPPOOLS summary displays a report of the HEAPPOOLS statistics and provides suggested
percentages for current cell sizes as well as suggested cell sizes.
• Specified Cell Size — the size of the cell specified in the HEAPPOOLS runtime option
• Element Size — the size of the cell plus any additional storage needed for control information or to
  maintain alignment
• Extent Percent — the cell pool percent specified by the HEAPPOOLS runtime option
• Cells Per Extent — the number of cells per extent. This number is calculated using the following formula, with a minimum of four cells:

\[
\text{Initial Heap Size} \times \frac{(\text{Extent Percent}/100))/\text{(Element Size)}}
\]

**Note:** Having a small number of cells per extent is not suggested because the pool can allocate many extents, which causes the HEAPPOLLS algorithm to perform inefficiently.

• Extents Allocated — the number of times that each pool allocated an extent.

To optimize storage usage, the extents allocated need to be either one or two. If the number of extents allocated is too high, increase the percentage for the pool.

• Maximum Cells Used — the maximum number of cells used for each pool.

• Cells In Use — the number of cells that were never freed.

A large number in this field can indicate a storage leak.

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

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

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

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

• Suggested Cell Sizes — sizes that are calculated to optimally use storage (assuming that the application will \texttt{malloc}/\texttt{free} with the same frequency).

**Note:** The suggested cell sizes are given with no percentages because the usage of each new cell pool size is not known. If there are less than 12 cell sizes calculated and the last calculated cell size is smaller than the largest cell size currently in effect, the largest cell size currently in effect is used for the last suggested cell size.

For more information about stack and heap storage, see *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*.

**C function \texttt{\_uheapreport()} storage report**

To generate a user-created heap storage report use the C function, \texttt{\_uheapreport()}. Use the information in the report to assist with tuning your application's use of the user-created heap.

For more information on the \texttt{\_uheapreport()} function, see *z/OS XL C/C++ Runtime Library Reference*. For tuning tips, see *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*. 
User-created HeapPools statistics

- Pool \( p \) size: \( sss\)
  - \( p \) — the number of the pool
  - \( sss \) — the cell size specified for the pool.
- Successful Get Heap requests: \( xxxx-yyyy n\)
  - \( xxxx \) — the low side of the range
  - \( yyyy \) — the high side of the range
  - \( n \) — the number of requests in the range.
- Requests greater than the largest cell size — the number of storage requests that are not satisfied by heap pools.

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

HeapPools summary

The HeapPools summary displays a report of the HeapPool statistics and provides suggested percentages for current cell sizes as well as suggested cell sizes. Figure 268 on page 467 shows a sample storage report generated by \texttt{__uheapreport()}.

- Cell Size — the size of the cell specified on the \texttt{__ucreate()} call
- Cells Per Extent — the cell pool count specified on the \texttt{__ucreate()} call
- Extents Allocated — the number of times that each pool allocated an extent in order to optimize storage usage.
- Maximum Cells Used — the maximum number of cells used for each pool.
- Cells In Use — the number of cells that were never freed.

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

- Suggested Cell Sizes — sizes that are calculated to optimally use storage (assuming that the application will \texttt{__umalloc/\_ufree} with the same frequency).
The suggested cell sizes are given with no cell counts because the usage of each new cell pool size is not known. If there are less than 12 cell sizes calculated, then the last pool size is set at 65536.
Appendix A. Diagnosing problems with Language Environment

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

Diagnosis checklist

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

1. If your failing application contains programs that were changed since they last ran successfully, review the output of the compile or assembly (listings) for any unresolved errors.
2. If there have not been any changes in your applications, check the output (job or console logs, CICS transient (CESE) queues) for any messages from the failing run.
3. Check the message prefix to identify the system or subsystem that issued the message. This can help you determine the cause of the problem. Following are some of the prefixes and their respective origins.
   - **EDC**
     The prefix for C/C++ messages. The following series of messages are from the C/C++ runtime component of Language Environment: 5000 (except for 5500, which are from the DSECT utility), 6000, and 7000.
   - **IGZ**
     The prefix for messages from the COBOL runtime component of Language Environment.
   - **FOR**
     The prefix for messages from the Fortran runtime component of Language Environment.
   - **IBM**
     The prefix for messages from the PL/I runtime component of Language Environment.
   - **CEE**
     The prefix for messages from the common runtime component of Language Environment.
4. For any messages received, check for recommendations in the “Programmer Response” sections of the messages in this information.
5. Verify that abends are caused by product failures and not by program errors. See the appropriate chapters in this manual for a list of Language Environment-related abend codes.
6. Your installation may have received an IBM Program Temporary Fix (PTF) for the problem. Verify that you have received all issued PFIs 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 33 or Chapter 12, “Using Language Environment AMODE 64 debugging facilities,” on page 337.

• 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” on page 470.

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

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

10. Record the conditions and options in effect at the time the problem occurred. Compile your program with the appropriate options to obtain an assembler listing and data map. If possible, obtain the binder or linkage editor output listing. Note any changes from the previous successful compilation or run. For an explanation of compiler options, see the compiler-specific programming guide.

11. If you are experiencing a no-response problem, try to force a dump. For example, CANCEL the program with the dump option.

12. Record the sequence of events that led to the error condition and any related programs or files. It is also helpful to record the service level of the compiler associated with the failing program.

Locating the name of the failing routine for a non-XPLINK application

If a system dump is taken, follow the save area chain to find out the name of the failing routine and whether IBM owns it. Following are the procedures for locating the name of the failing routine, which is the primary entry point name.

1. Find the entry point associated with the current save area. The entry point address (EPA), located in the previous save area at displacement X'10', decimal 16, points to it.

2. Determine the entry point type, of which there are four:

<table>
<thead>
<tr>
<th>Entry point type is...</th>
<th>If...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language Environment conforming</td>
<td>The entry point plus 4 is X'00C3C5C5'.</td>
</tr>
<tr>
<td>Language Environment conforming OPLINK</td>
<td>The entry point plus 4 is X'01C3C5C5'. OPLINK linkage conventions are used.</td>
</tr>
<tr>
<td>C/C++</td>
<td>The entry point plus 5 is X'CE'.</td>
</tr>
<tr>
<td>Nonconforming</td>
<td>The entry point is none of the above. Nonconforming entry points are for routines that follow the linking convention in which the name is at the beginning of the routine. X'47F0Fxxx' is the instruction to branch around the routine name.</td>
</tr>
</tbody>
</table>

For routines with Language Environment-conforming and C/C++ entry points, Language Environment provides program prolog areas (PPAs). PPA1 contains the entry point name and the address of the PPA2; PPA2 contains pointers to the timestamp, where release level keyword information is found, and to the PPA1 associated with the primary entry point of the routine.

• If the entry point type of the failing routine is Language Environment-conforming, go to step “3” on page 471.

• If the entry point type is C/C++, go to step “5” on page 471.

• If the entry point type is nonconforming, go to step “6” on page 472.
3. If the entry point type is Language Environment-conforming, find the entry point name for the Language Environment or COBOL program.
   a. Use an offset of X'C' from the entry point to locate the address of the PPA1.
   b. In the PPA1, locate the offset to the length of the name. If OPLINK, then multiply the offset by 2 to locate the actual offset to the length of the name.
      
      **Note:** Enterprise COBOL V5.1 and later releases use OPLINK.
   c. Add this offset to the PPA1 address to find the halfword containing the length of the name, followed by the entry point name.

      The entry point name appears in EBCDIC, with the translated version in the right-hand margin of the system dump.

4. Find the Language Environment or COBOL program name.
   a. Find the address of PPA2 at X'04' from the start of PPA1. For Enterprise COBOL V5.1 or later releases, find a signed offset at X'04' from the start of PPA1, then add this offset to the entry point address to obtain the address of PPA2.
   b. Find the address of the compilation unit's primary entry point at X'10' in the PPA2. For Enterprise COBOL V5.1 and later releases, find a signed offset at X'10' in the PPA2, then add this offset to the address of PPA2 to obtain the compilation unit's primary entry point.
   c. Find the entry point name associated with the primary entry point as described above. The primary entry point name is the routine name.

See *z/OS Language Environment Vendor Interfaces* for details of:
- the non-XPLINK Language Environment-conforming PPA1 and PPA2.
- the XPLINK Language Environment-conforming PPA1, and the XPLINK PPA1 optional area fields.
- the non-XPLINK Language Environment PPA2.
- the Language Environment PPA2: Compile Unit Block for XPLINK.
- the PPA2 timestamp and version information.

5. If the entry point type is C/C++, find the C/C++ routine name.
   a. Use the entry point plus 4 to locate the offset to the entry point name in the PPA1 (see Figure 269 on page 472).
   b. Use this offset to find the length-of-name byte followed by the routine name.

      The routine name appears in EBCDIC, with the translated version in the right-hand margin.
6. If the entry point type is nonconforming, find the PL/I routine name.
   a. Find the one byte length immediately preceding the entry point. This is the length of the routine
      name.
   b. Go back the number of bytes specified in the name length. This is the beginning of the routine
      name.

7. If the entry point type is nonconforming, find the name of the routine other than PL/I.
   a. Use the entry point plus 4 as the location of the entry point name.
   b. Use the next byte as the length of the name. The name directly follows the length of name byte. The
      entry point name appears in EBCDIC with the translated version in the right-hand margin.

Figure 270 on page 472 shows a nonconforming entry point type. Nonconforming entry points that can
appear do not necessarily follow this linking convention. The location of data in these save areas can
be unpredictable.

---

**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)” on page 473.

Preparing documentation for an authorized program analysis report (APAR)

This section provides an overview of how to prepare documentation if a problem arises. For more information, see the Software Support Handbook (www.ibm.com/support/customercare/sas/f/handbook/home.html).

Follow these steps before you prepare documentation for an APAR:

• 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.

After you meet these criteria, follow these instructions:

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 they consult 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 that 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 that is described in the PMR is a problem with the Language Environment product, they will work with you to open an APAR, so the problem can be fixed.

2. Provide APAR documentation. When you submit an APAR, you will need to supply information that describes the failure. Table 69 on page 474 describes how to produce documentation that is required for submission with the APAR.
Table 69: Problem resolution documentation requirements

<table>
<thead>
<tr>
<th>Item</th>
<th>Materials required</th>
<th>How to obtain materials</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Machine-readable source program, including macros, subroutines, input files, and any other data that might help to reproduce the problem.</td>
<td>IBM-supplied system utility program</td>
</tr>
<tr>
<td>2</td>
<td>Compiler listings:</td>
<td>Use appropriate compiler options</td>
</tr>
<tr>
<td></td>
<td>• Source listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Object listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Storage map</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Traceback</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Cross-reference listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• JCL listing and linkage editor listing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Assembler-language expansion</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Dumps</td>
<td>See instructions in Chapter 3, “Using Language Environment debugging facilities,” on page 33 (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 that are used to invoke and run the routine, including all runtime options, in machine-readable form</td>
<td>Application programmer</td>
</tr>
<tr>
<td>8</td>
<td>System output that is associated with the MSGFILE runtime option.</td>
<td>Specify MSGFILE(SYSOUT)</td>
</tr>
<tr>
<td>9</td>
<td>Contents of the applicable catalog</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A hardcopy log of the events leading up to the failure.</td>
<td>Print each display.</td>
</tr>
</tbody>
</table>

3. Submit the APAR documentation.

When you submit 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 that is submitted must have the following information attached and visible:

- The APAR number that is 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 following information:
  - The exact JCL listing or the list of commands that are 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 that is 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 that is used for each data set.

Any printed materials must show the corresponding APAR number.
The IBM service personnel will inform you of the mailing address of the service center nearest you.

If an electronic link with IBM Service is available, use this link to send diagnostic information to IBM Service.

After the APAR is opened and the fix is produced, the description of the problem and the fix will be in the software support database in SIS, accessible through ServiceLink.
Appendix B. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed email message to mhvrcfs@us.ibm.com.

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- z/OS TSO/E Primer
- z/OS TSO/E User's Guide
- z/OS ISPF User's Guide Vol I

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that the screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.
Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 \* FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* \* FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element

The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element

The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE (KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

* indicates an optional syntax element that is repeatable

The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3* , 3 HOST, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

**+ indicates a syntax element that must be included**

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.
This information was developed for products and services that are offered in the USA or elsewhere.

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

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

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

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

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

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

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for the Knowledge Centers. Use of hyperlinks in other output formats of this information is at your own risk.

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

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Site Counsel
2455 South Road
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

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

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

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

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

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or
reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

**Rights**

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

**IBM Online Privacy Statement**

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user’s name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at ibm.com/privacy and IBM’s Online Privacy Statement at ibm.com/privacy/details in the section entitled “Cookies, Web Beacons and Other Technologies,” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at ibm.com/software/info/product-privacy.

**Policy for unsupported hardware**

Various z/OS elements, such as DFSMS, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.
Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This publication documents information NOT intended to be used as a Programming Interface of Language Environment in z/OS.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and Trademark information (www.ibm.com/legal/copytrade.shtml).

Adobe, Acrobat, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

Other company, product, and service names might be trademarks or service marks of others.
Index

Special Characters

__abend 164, 430
__alloc 164, 430
__amrc 164, 430
__cabend() function 329, 336, 354
__code 164, 430
__error 164, 430
__feedback 164, 430
__last_op 164, 430
__le_cib_get() function 329
__le_message_get_and_and_write() function 324, 331
__le_message_get() function 331
__le_msg_write() function 324, 331
__msg 165, 431
__reset_exception_handler() function 329
__set_exception_handler() function 329, 330
_BPXK_MDUMP 83, 355
_EDC_ADD_ERRNO2 169, 436

Numerics

64-bit applications 323, 333, 337, 429

A

abend codes
  < 4000 28, 334
  >= 4000 28, 334
  4093 333
  passing to operating system 23
  system, example of 30, 335
  user-specified, example of 30, 335
  user, example of 30, 335, 336
  using 30, 335

abends
  internal, table of output 318
  Language Environment 30, 315, 335
  requested by assembler user exit 23
  system 30, 336
  under CICS 315
  user 30, 336

ABPERC runtime option
  function 8
  generating a system dump and 81
  modifying condition handling behavior and 21

ABTERMENC runtime option
  using 23

accessibility
  contact IBM 477
  features 477

AGGREGATE compiler option 4, 7

AMODE 64 applications
  classifying errors 333
  debugging C/C++ 429
  preparing for debugging 323
  using debugging 337

anywhere heap
  statistics 18

APAR (Authorized Program Analysis Report)
documentation 473

application program interfaces (API) 329, 331

application programs
  debugging
    handling a storage dump written to a BFS file 213,
    462
    handling a storage dump written to an HFS file 213,
    462

argument
  in dump 62

arguments, registers, and variables for active routines 60,
  353

assembler language
  user exit
    for CICS 317
    generating a system dump with 81, 354
    modifying condition handling behavior and 23
    using 22, 23

assistive technologies 477

atexit
  information in dump 187

automatic variables
  locating in dump 274, 300

B

base locator
  for working storage 230
  in dump 230

below heap
  statistics 18

BLOCKS option of CEE3DMP callable service 35

C

C library function
  trace table entries for 448

C return codes to CICS 315

C/C++
  __amrc
    example of structure 164, 430
    information in dump 187
  __msg 165, 431
  atexit
    information in dump 187
  cdump() function 178, 440
  compiler options 3
  debugging examples 203, 210, 451
  dump
    information in 187
    parameter in 175
    structure variables, locating in 176
    system, structures in 176
    file
C/C++ (continued)

file (continued)
control block information 187
status and attributes in dump 187
functions
calling dump, example 178, 440
cdump() 40, 178, 339, 440
csnap() 40, 178, 179, 339, 440, 441
ctrace() 40, 178, 179, 339, 440, 441
fetch() 163, 429
fopen() 165, 431
perror() 163, 169, 429, 435
printf() 164, 430
to produce dump output 40, 339
memory file control block 188
stdio.h 164, 430
timestamp 178
CAA (common anchor area) 64, 353, 358, 360, 372, 404
call chain 62
CALL statement
CDUMP/CPDUMP 249
DUMP/PDUMP 247
SDUMP 249
callable services 20
case 1 condition token 25, 331
case 2 condition token 25, 331
cdump() function 178, 334, 440
CEE prefix 27, 29, 333, 335
CEE3ABD—terminate enclave with an abend
generating a dump and 81
handling user abends and 20, 30
modifying condition handling behavior and 20
CEE3DMP—generate dump
generating a Language Environment dump with 33
options 34
relationship to PLIDUMP 268, 296
syntax 34
CEE3GRO—returns location offset 20
CEE3SRP—set resume point 20
CEEBXITA assembler user exit 22, 23
CEEEXITA assembler user exit 317
CEEDCOD—decompose a condition token 25
CEEDUMP—Language Environment Dump Service
control blocks 104, 371
locating 151
CEEDUMPs, controlling access 149
CEEHDLR—register user condition handler 22
CEEHDLR—register user exception handler 330
CEEMGET—get a message 25
CEEMOUT—dispatch a message 24
CEEMRCE—move resume cursor to designated label 20
CEEMRCR—move resume cursor relative to handle cursor 20
CEEMSG—get, format, and dispatch a message 25
CEESGL—signal a condition 25
CEESTART 163
CEL prefix 333
CELQSTAT 429
character
data dump 249
CHECK runtime option
function 8
modifying condition handling behavior and 21
CHECKOUT compiler option 4
CICS

CICS (continued)
abends
application, from an EXEC CICS command 318, 319
debugging for 313
debugging information, table of locations 313
destination control table (DCT) 313
example traceback in CESE transient data queue 313
examples of output 313
nonzero reason code returned, table of output 318
reason codes 315
register and program status word contents 314
return codes
Language Environment 315
runtime messages 313
transaction
dump 314
rollback 317
class test 224
classifying errors table 27, 333
CLLE (COBOL load list entry) 229
COBCOM control block 233
COBOL
base locator for working storage 230
C-specific common anchor area (C-CAA) 186
compiler options 5
debugging examples 233, 239
dump
external data in 230
file information in 230
linkage section in 230
local variables in 227
routine information in 227
run unit storage in 232
signal information in 186
stack frames for active routines in 227
working storage in 230
errors 223
listings 225
memory file control block 186
program class storage 230
return codes to CICS 315
routine
calling Language Environment dump service 226
COBVEC control block 233
command
syntax diagrams xxii
COMMAREA (Communication Area) 314
compiler options
C 3
COBOL 5
Fortran 6
LP64 323, 429
PL/I 7
condition
information
for active routines 59, 353
in dump 59, 353
POSIX 25
unhandled 26, 332
condition handling
behavior, modifying 20
user-written condition handler 20, 22
condition information block (CIB) 62, 75
condition manager 26, 332
CONDITION option of CEE3DMP callable service 36, 62
condition token
  case 1 25, 331
  case 2 25, 331
  example of 26, 331
  conditions, nested 26, 332
  contact
    z/OS 477
control block
  for active routines 60, 353
  csnap() function 179, 441
ctrace() function 179, 334, 441

D
data
  map listing 226
  values 62

DCB (data control block) 229
DCT (destination control table) 313
DEBUG runtime option 8
dumping
  C, examples 203, 210, 451
  COBOL, examples 233, 239
  Fortran, examples 254, 258
  PL/I, examples 277, 283, 303, 307
tool 33, 337
DEPTHCONDLMT runtime option
  function 8, 9, 323
  modifying condition handling behavior and 21
  wait/loop error and 28
diagnosis checklist 469
display
  errnojr_value 31
DISPLAY statement 24, 223
diagrams
  errnojr_value 31
divide-by-zero error 203, 239, 255, 283, 286, 307, 311, 451
DSA (dynamic save area) 62
dummy DSA 62
dump
  an area of storage 249
  date in 55, 350
  dynamically allocated storage in 61
  storage
    written to an HFS file 213, 462
  symbolic 249
dumping
  DUMP suboption of TERMTHDACT runtime option 36, 337
DUMP/PDUMP routine
  format specifications 248
  output 248
  usage considerations 248
DYNDUMPs, controlling access 149

E
ECB (enclave control block) 61, 354
EDB (enclave data block) 61, 354
EDC prefix 27, 29, 333, 335
edcmtext shell command 31
EIB (exec interface block) 314
enclave (continued)
  identifier in dump 56, 350
  member list 55–62, 354
  storage 61
  termination
    behavior, establishing 23
diagnosis checklist 469
entry information 55, 350
ENTRY option of CEE3DMP callable service 36
entry point
  name of active routines in dump 351
ERRCOUNT runtime option
  function 8, 9
  modifying condition handling behavior and 21
  wait/loop error and 28
erro 186
errojr_value
  displaying 31
error
  determining source of 469
  message while Language Environment was handling
  another error 26, 332
  unanticipated 333
ESD compiler option 7
dumping
  application abends from 318, 319
  C routines 203, 210, 451
  calling a nonexistent subroutine 235, 281, 306
  COBOL routines 233, 239
  divide-by-zero error 203, 239, 255, 283, 286, 307, 311, 451
  Fortran routines 254, 258
  output under CICS 313
  PL/I routines 277, 283, 303, 307
  SUBSCRIPTRANGE error 233, 277, 303
exception handling
  behavior, modifying 329
  user-written exception handler 329
EXEC CICS DUMP statements 314
external data
  for COBOL programs in dump 230

F
feedback xxv
dumping
  fetch information in dump 187
  fetch() function 163, 429
dumping
  fetchable module 163, 429
file
  for COBOL, in dump 230
  status key 223
file control block (FCB) 187
FILES option of CEE3DMP callable service 35
FLAG compiler option 4
floating point registers
  in dump 56
fopen() function 165, 431
FOR prefix 27, 29
Fortran
  compiler options 6
dumping
  debugging examples 254, 258
dumping
  dump services 247
dumping
  errors, determining the source of 245
listings 246
G

general purpose registers 56
GMAREA 229
GONUMBER compiler option 4

H

HANDLE ABEND EXEC CICS command 313
header files, C
cctest.h 178, 440
ererro.h 203, 451
stdio.h 164, 430
stdlib.h 203, 451
heap pool	race report 358
heap pools	race report 115, 390
heap storage
created by CEECRHP callable service 19
in LEDATA Output
reports 109, 382
storage in dump 61
user 16
HEAP64 runtime option 326
HEAPCHK runtime option
function 9, 111, 323, 384
HeapPools
storage statistics 467
HEAPPOOLS
storage statistics
user-created. __uheapreport() 466
user-created. _uheapreport 216
trace report 86, 115
HEAPPOOLS runtime option 326
HEAPPOOLS64
storage statistics 464
HEAPPOOLS64 runtime option 326
HEAPZONE runtime option
function 9
HEAPZONES runtime option 324

I

I/O

conventions 163, 429
IBM prefix 27, 29
IGZ prefix 27, 29
INFOMSGFILTER runtime option
function 9, 324
INITIALIZE statement 224
instance specific information (ISI) 332
instruction length counter (ILC) in dump 59, 353
Inter-procedural Analysis (IPA) 323
interactive problem control system (IPCS)
analyzing a storage dump 213, 462
cbf command 417
VERBEXIT 356, 357, 360, 394, 406
INTERRUPT compiler option
function 7
INTERRUPT runtime option 8, 9
interruption code in dump 59, 353
IOHEAP64 runtime option 326

ITBLK in dump 233

K

keyboard
navigation 477
PF keys 477
shortcut keys 477

L

LAA (library anchor area) 418
language constructs 223
Language Environment
ccontent, new xxvi
return codes to CICS 315
summary of changes for V2R1 as updated February 2015 xxvii
summary of changes for V2R3 xxvi, xxvii
symbolic feedback code 25, 331
Language Environment dump
C information in 186
CEEDUMP 8, 33, 323
COBOL information in 228
default options 34
example traceback in 61, 354
Fortran information in 252
multiple enclaves and 79
options
BLOCKS 35
CONDITION 36, 62
ENCLAVE 34
ENTRY 36
FILES 35
FNAME 35
NOBLOCKS 35
NOCONDITION 36
NOENTRY 36
NOFILES 35
NOSTORAGE 35
NOTRACEBACK 35
NOVARIABLES 35
Pagesize(n) 35
STACKFRAME 35
STORAGE 35
THREAD 35
TRACEBACK 35, 62
VARIABLES 35, 62
output
for C routines 182
for COBOL program 225
for Fortran routines 252
for PL/I routines 268, 296
information for multiple enclaves 40, 339
PL/I information in 270, 298
section descriptions 54, 350
TERMTHDACT suboptions 38, 338
title 55, 350
traceback with condition information
C routine 181, 444
COBOL program 227
Fortran routine 245, 252, 253
Language Environment routine 54, 350
Language Environment dump (continued)  
traceback with condition information (continued)  
PL/I routine 268, 296  
using C functions 40, 339  
using CDUMP/CPDUMP subroutine 247  
using CEE3DMP callable service 33, 54, 350  
using DUMP/PDUMP subroutine 247  
using PLIDUMP subroutine 40, 268, 296  
using SDUMP subroutine 247  
using TERMTHDACT runtime option 36, 337  
LCA (library communication area) 418  
LEDATA  
IPCS Verbexit  
C/C++ Output 118  
COBOL Output 138  
Parameters 85  
Understanding Output 88  
IPCS VERBEXIT  
C/C++ Output 394  
Parameters 357  
PL/I Output 406  
Understanding Output 360  
LIBHEAP64 runtime option 326  
linkage section  
for COBOL programs in dump 230  
LIST compiler option 4, 5, 7  
largest generated by compiler  
COBOL 225  
Fortran 246  
PL/I 262, 290  
LMESSAGE compiler option 7  
local  
variables 60  
LP64  
compiler option 323, 429  

M  
machine state information  
in dump 59, 353  
MAP compiler option 5, 7  
MEMCHECK VHM memory leak analysis tool 218  
MEMLIMIT storage parameter 333  
memory file control block (MFCB) 186, 188  
message  
classifying errors and 28, 334  
runtime, CICS 313  
user-created 24  
using in your routine 24  
module  
fetchable 163, 429  
module name prefixes, Language Environment 27, 333  
MSG suboption  
of TERMTHDACT 36, 337  
MSGFILE runtime option  
function 9  
runtime messages and 28, 334  
MSGQ runtime option 9  

N  
navigation  
keyboard 477  
nested condition 26, 332  
no response (wait/loop) 28, 334  
NOBLOCKS option of CEE3DMP callable service 35  
NOCONDITION option of CEE3DMP callable service 36  
NOENTRY option of CEE3DMP callable service 36  
NOFILES option of CEE3DMP callable service 35  
NOSTORAGE option of CEE3DMP callable service 35  
NOTRACEBACK option of CEE3DMP callable service 35  
NOVARIABLES option of CEE3DMP callable service 35  

O  
OFFSET compiler option 4, 5, 7  
optimizing  
C 4, 62  
COBOL 5  
Fortran 250  
PL/I 7  
options  
C compiler 3  
COBOL compiler 5  
defaults for dump 37, 338  
determining runtime in effect 9, 11, 324  
Fortran compiler 6  
Language Environment runtime 8, 323  
PL/I compiler 7, 8  
out-of-storage condition  
virtual storage 333  
OUTDD compiler option 5  
output  
incorrect 28, 334  
missing 28, 334  

P  
page number  
in dump 55, 350  
PAGESIZE(n) option of CEE3DMP callable service 35  
parameter  
checking value of 23  
PCB (process-level control block) 411  
perror() function 169  
perror() function 435  
PL/I  
address of interrupt, finding in dump 272, 299  
CAA address, finding in dump 276, 302  
common anchor area (CAA) 276, 302  
compiler listings  
object code listing 266  
static storage map 265  
variable storage map 266  
compiler options  
generating listings with 262, 290  
list of 7  
CSECT 265  
debugging examples 277, 283, 303, 307  
dump  
error type, finding in 272, 299  
parameter list, finding contents in 275, 301  
PL/I information, finding in 270, 275, 298, 301  
PLIDUMP subroutine and 268, 296  
statement number, finding in 272, 299  
timestamp, finding in 275  

489
PL/I (continued)
dump (continued)
  variables, finding in 274, 300
  ERROR ON-unit 260, 272, 288, 299
  errors 259, 261, 287, 289
  floating-point register 260, 288
  object code listing 266
  ON statement control block 266
  static storage listing 265
  SUBSCRIPTRANGE condition 260, 261, 277, 281, 288, 289, 303, 305
PLIDUMP subroutine 269, 296
PMR (problem management record) 473
pointer
  variable 163, 429
PPA 470
preventive service planning (PSP) bucket 469
printf() function 24, 164, 430
problem management record (PMR) 473
procedure division listings 226
process
  control block 61, 354
  member list 55–62, 354
process control block (PCB) 55–62, 354
PROFILE runtime option
  function 9, 324
program
  class storage 230
  program prolog area 470
  program status word (PSW) 59, 353
PSP (preventive service planning) bucket 469
Q
QUIET suboption of TERMTHDACT runtime option 36, 337
R
reason code
  nonzero returned to CICS 318
  under CICS 315
registers 0–15
  in dump 56
release number
  in dump 55, 350
request parameter list (RPL) 187
return code
  bad or nonzero 28, 334
routines
  common errors in 27
RPTOPTS runtime option 10
RPTSTG runtime option 11, 326
run unit
  COBOL 232
  level control block 232
runtime messages
  under CICS 313
runtime options
  determining those in effect 9, 11, 324
  sample options report 10
  specifying 23
S
scope
terminator 223
SDUMP routine
  description 249
  format specifications 250
  output 249
  usage considerations 250
sending to IBM
  reader comments xxv
service routines
  CDUMP/CPDUMP 249
  DUMP/PDUMP 247
  SDUMP 249
SET statement 224
shortcut keys 477
signal information in dump 186
sorted cross-reference listing 226
SOURCE compiler option 4, 5, 7
stack
  frame 62
  frame format 62, 63
STACK64 runtime option 326
STACKFRAME option of CEE3DMP callable service 35
statement numbers
  in dump 55, 350
static
  variables in dump 274, 300
  writable map 173, 176, 177
status
  of routines in dump 351
stderr 24, 334, 337, 338
stdio.h 164, 430
stdout 24
storage
  evaluating use of 11, 326
  for active routines 60, 353
  leak detecting 61, 111, 354, 384
  report 11, 326
  statistics 16, 18
STORAGE compiler option 7
storage dump
  written to an HFS file 213, 462
STORAGE option of CEE3DMP callable service 35
STORAGE runtime option 9, 324
structure
  map 176
  variable example code 176
summary of changes
Language Environment
  V2R1 as updated February 2015 xxvii
  V2R3 xxvi, xxvii
Summary of changes xxvii
symbolic
  feedback code 25, 331
  symbolic dumps
    how to call under Fortran 249
  symbolic feedback code
    using the 26
syntax diagrams
  how to read xxii
system abend
  with TRAP(OFF) 28, 334
system abend (continued)
  with TRAP(ON) 28, 334
system dump
  generating
    in z/OS UNIX shell 83

T
task global table (TGT) 229
TERMINAL compiler option 4, 7
TERMTHDACT runtime option
  function 8, 36, 281, 306, 323, 337
  generating a dump and 21, 330
  modifying condition handling behavior and 9, 324
  suboptions 36, 337
TEST compiler option 4, 5, 7
TEST runtime option 9, 324
text file name prefixes, Language Environment 27, 333
THDCOM in dump 233
THREAD option of CEE3DMP callable service 35
THREADSTACK64 runtime option 326
time
  in dump 55, 350
TRACE runtime option
  function 9, 324
  trace table 152, 420
TRACE suboption of TERMTHDACT runtime option 36, 337
TRACEBACK option of CEE3DMP callable service 35, 62
transaction
  dump 314
  rollback 317
  rollback effects of assembler user exit on 317
  work area 314
TRAP runtime option
  function 9, 324
  Language Environment condition handling and 21, 81, 354
  Language Environment exception handling and 330

U
UADUMP suboption of TERMTHDACT runtime option 21, 37, 330, 338
UAIMM suboption of TERMTHDACT runtime option 21, 37, 330, 338
UAONLY suboption of TERMTHDACT runtime option 21, 37, 330, 337
UATRACE suboption of TERMTHDACT runtime option 21, 37, 330, 338
unhandled conditions
  establishing enclave termination behavior for 23
USE EXCEPTION/ERROR declaratives 224
USE FOR DEBUGGING declarative 224
user
  abend
    code 28, 334
  exit 23
  heap
    statistics 18
  stack
    statistics 16
user interface (continued)
  TSO/E 477
user-specified abends 30, 335
USRHDLR runtime option 8, 22, 323
utility and service subroutines
  CDUMP/CPDUMP 249
  DUMP/PDUMP 247
  SDUMP 249

V
variables
  in Language Environment dump 62
  structure example code 176
VARIABLES option of CEE3DMP callable service 35, 62
VBREF compiler option 6
VERBEXIT
  LEDATA 85, 356
  version number
    in dump 55, 350

W
working storage
  in dump 60, 230, 353
Writable Static Area 172

X
XPLINK
  downward-growing stack 63
  finding XPLINK information in a dump 195
  stack frame format 62, 63
  storage statistics 17
  trace table entries for 195
XREF compiler option 6, 7
XUFLOW runtime option
  function 9
  modifying condition handling behavior and 22

Z
z/OS UNIX System Services
  C application program and 213, 462
  generating a system dump 83