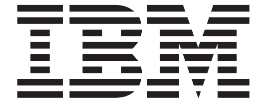
IBM Language Environment for VSE/ESA

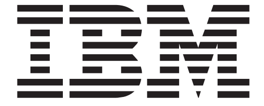


Programming Guide

Version 1 Release 4 Modification Level 4

SC33-6684-04

IBM Language Environment for VSE/ESA



Programming Guide

Version 1 Release 4 Modification Level 4

SC33-6684-04

Note!
Before using this information and the product it supports, be sure to read the general information under "Notices" on page xiii.

Fifth Edition (March 2005)

This edition applies to Version 1 Release 4 Modification Level 4 of IBM Language Environment for VSE/ESA, 5686-CF7, and to any subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the addresses given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Deutschland Entwicklung GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

You may also send your comments by FAX or via the Internet:

Internet: s390id@de.ibm.com
FAX (Germany): 07031-16-3456
FAX (other countries): (+49)+7031-16-3456

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

© Copyright International Business Machines Corporation 1991, 2005. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures . . . ix

Tables . . . xi

Notices . . . xiii
Programming Interface Information . . . xiii
Trademarks and Service Marks . . . xiv

About This Book . . . xv
What Is LE/VSE? . . . xv
LE/VSE-Conforming Languages . . . xvi
  LE/VSE Compatibility with Previous Versions of COBOL . . . xvi
Terms Used in This Book . . . xvii
How This Book Is Organized . . . xvii
How to Read the Syntax Diagrams . . . xvii

Where to Find More Information . . . xxi
Softcopy Publications . . . xxiii

Summary of Changes . . . xxv
Changes Introduced with Fifth Edition (March 2005) . . . xxv
Changes Introduced with Fourth Edition (March 2003) . . . xxv
Changes Introduced with Third Edition (December 2001) . . . xxvi

Part 1. Linking and Running Applications with LE/VSE . . . 1

Chapter 1. Introduction to LE/VSE . . . 3

Chapter 2. Preparing to Link and Run under LE/VSE . . . 5
Understanding the Basics . . . 5
  Planning to Link and Run . . . 5
  Link-Editing Single-Language Applications . . . 6
  Link-Editing ILC Applications . . . 6
Checking Which Run-Time Options Are in Effect . . . 6
COBOL Considerations . . . 7
  COBOL Linking Considerations . . . 7
PL/I Linking Considerations . . . 7
  Link-Editing Fetchable Phases . . . 7
  Fetching Phases with Different AMODEs . . . 8
C AMODE/RMODE Considerations . . . 9

Chapter 3. Prelinking an Application . . . 11
Which Programs Need to Be Prelinked . . . 11
What the Prelinker Does . . . 11
Prelinking Process . . . 11
  Using the Prelinker Automatic Library Call . . . 12
LE/VSE Prelinker Map . . . 13
Control Statement Processing . . . 15
  INCLUDE Control Statement . . . 15
  LIBRARY Control Statement . . . 16
  RENAME Control Statement . . . 16
Mapping L-Names to S-Names . . . 18
Running the Prelinker . . . 19
Prelinker Options . . . 19

Chapter 4. Linking and Running . . . 21
Basic Linking and Running . . . 21
  Linking and Running an Existing Object Module 21
  Overriding the Default Run-Time Options . . . 21
Providing Input to the Linkage Editor . . . 22
  Writing JCL for the Linkage Editor . . . 23
  Link-Editing Multiple Object Modules . . . 26
  Using the INCLUDE Statement . . . 26
  Linkage Editor Module Map . . . 26
  Detecting Link-Edit Errors . . . 28
Running an Application . . . 29
  Specifying the Search Order . . . 29
Specifying Run-Time Options . . . 30
  Specifying Run-Time Options in the EXEC Statement . . . 30
Using the iconv Utility for C . . . 31
Using the genxlt Utility for C . . . 31

Chapter 5. Using Run-Time Options . . . 33
Understanding the Basics . . . 33
Specifying Run-time Options . . . 35
  Order of Precedence . . . 37
  Specifying Suboptions in Run-Time Options . . 38
  Specifying Run-Time Options and Program Arguments . . . 38
  C Compatibility Considerations . . . 39
  COBOL Compatibility Considerations . . . 39
  PL/I Compatibility Considerations . . . 39
  CEEXOPT Invocation Syntax . . . 40
  Notes on CEEXOPT Invocation . . . 44
  Performance Considerations . . . 45
  Printing CICS-Wide Run-Time Options to Console . . . 45

Part 2. Preparing an Application to Run with LE/VSE . . . 47

Chapter 6. Using LE/VSE Parameter List Formats . . . 49
Understanding the Basics . . . 49
Argument Lists and Parameter Lists . . . 50
Passing Arguments between Routines . . . 50
Preparing Your Main Routine to Receive Parameters 51
  PL/I Argument-Passing Considerations . . . 55

© Copyright IBM Corp. 1991, 2005

Chapter 7. Routines That Must Be Reentrant . . . 57
Understanding the Basics . . . 57
Making Your C Program Reentrant . . . 57
  Natural Reentrancy . . . 57
  Constructed Reentrancy . . . 57
  Generating a Reentrant C Object Module . . . 58
Making Your COBOL Routine Reentrant . . . 58
Making Your PL/I Routine Reentrant . . . 58
Installing a Reentrant Phase . . . 58

Part 3. Concepts, Services, and Models . . . 59

Chapter 8. Initialization and Termination Under LE/VSE . . . 63
Understanding the Basics . . . 63
LE/VSE Initialization . . . 65
  What Happens During Initialization . . . 65
LE/VSE Termination . . . 66
  What Causes Termination . . . 66
  What Happens During Termination . . . 67
Managing Return Codes in LE/VSE . . . 68
  How the LE/VSE Enclave Return Code is Calculated . . . 68
  Setting and Altering User Return Codes . . . 69
Termination Behavior for Unhandled Conditions . . . 70
  Determining the Abend Code . . . 71

Chapter 9. Program Management Model 75
Understanding the Basics . . . 75
  Program Management Model Terminology . . . 75
  Processes . . . 77
  Enclaves . . . 77
  Threads . . . 79
The Full Language Environment Program Management Model . . . 79

Chapter 10. Stack and Heap Storage . . . 81
Understanding the Basics . . . 81
Stack Storage Overview . . . 83
  Tuning Stack Storage . . . 84
  COBOL Considerations . . . 84
  PL/I Storage Considerations . . . 84
Heap Storage Overview . . . 85
  Heap IDs Recognized by the LE/VSE Heap Manager . . . 86
  AMODE Considerations for Heap Storage . . . 87
  Tuning Heap Storage . . . 87
  COBOL Considerations . . . 87
Storage Performance Considerations . . . 87
COBOL and LE/VSE Storage Considerations . . . 87
Dynamic Storage Services . . . 89
Examples of Callable Storage Services . . . 90
  C Example of Building a Linked List . . . 90
  COBOL Example of Building a Linked List . . . 92
  PL/I Example of Building a Linked List . . . 94
  C Example of Storage Management . . . 96
  COBOL Example of Storage Management . . . 98
  PL/I Example of Storage Management . . . 100

Chapter 11. LE/VSE Condition Handling Introduction . . . 103
Understanding the Basics . . . 103
  Related Run-Time Options and Callable Services 104
The Stack Frame Model . . . 105
  The Handle Cursor . . . 106
  The Resume Cursor . . . 106
What Is a Condition in LE/VSE? . . . 106
Steps in Condition Handling . . . 107
  Enablement Step . . . 107
  Condition Step . . . 109
  Termination Imminent Step . . . 111
Invoking Condition Handlers . . . 114
Responses to Conditions . . . 116
Condition Handling Scenarios . . . 116
  Scenario 1: Simple Condition Handling . . . 116
  Scenario 2: Condition Handling with User-Written Condition Handler Present for T_I_U . . . 118
  Scenario 3: Condition Handling with User-Written Condition Handler Present for Divide-by-Zero Condition . . . 119

Chapter 12. LE/VSE and HLL Condition Handling Interactions . . . 121
Understanding the Basics . . . 121
C Condition Handling Semantics . . . 121
  Comparison of C-LE/VSE Terminology . . . 122
  Controlling Condition Handling in C . . . 122
  C Condition Handling Actions . . . 124
  C Signal Representation of S/370 Exceptions 127
COBOL Condition Handling Semantics . . . 128
  COBOL Condition Handling Examples . . . 129
  Restrictions about Resuming Execution after an IGZ Condition Occurs . . . 131
  Reentering COBOL Programs after Stack Frame Collapse . . . 131
  Handling Fixed-Point and Decimal Overflow Conditions . . . 132
PL/I Condition Handling Semantics . . . 132
  PL/I Condition Handling Actions . . . 132
  Promoting Conditions to the PL/I ERROR Condition . . . 133
  Mapping Non-PL/I Conditions to PL/I Conditions . . . 134
  Additional PL/I Condition Handling Considerations . . . 134
  PL/I Condition Handling Example . . . 135

Chapter 13. Coding a User-Written Condition Handler . . . 137
Understanding the Basics . . . 137
Types of Conditions You Can Handle . . . 137
User-Written Condition Handler Interface using CEEHDLR . . . 138
Registering a User-Written Condition Handler using USRHDLR . . . 139
Nested Conditions . . . 140

Nested Conditions in Applications Containing a COBOL Program 140
Using LE/VSE Condition Handling with Nested COBOL Programs 141
Examples with a Registered User-Written Condition Handler 141
Handling a Divide-by-Zero Condition in C or COBOL 141
Handling an Out-of-Storage Condition in C or COBOL 148
Signaling and Handling a Condition in a C Routine 158
Handling a Divide-by-Zero Condition in a COBOL Program 160
Handling a Program Check in an Assembler Routine 165

Chapter 14. Using Condition Tokens 171
Understanding the Basics 171
Understanding the Structure of the Condition Token 172
The Effect of Coding the fc Parameter 173
Testing a Condition Token for Success 174
Testing Condition Tokens for Equivalence . . . 174
Testing Condition Tokens for Equality 175
The Effect of Omitting the fc Parameter 175
Using Symbolic Feedback Codes 175
Locating Symbolic Feedback Codes for Conditions 175
Including Symbolic Feedback Code Files . . . 176
Examples Using Symbolic Feedback Codes . . 178
Condition Tokens for C Signals under C 182
LE/VSE-provided q_data Structure for Abends . . 182

Chapter 15. Using and Handling Messages 185
Understanding the Basics 185
Creating Messages. 185
Creating a Message Source File 186
Using the CEEBLDTX Utility 189
Files Created by CEEBLDTX 190
Running the CEEBLDTX Utility 191
Assembling and Link-Editing the Message File 192
CEEBLDTX Error Messages 192
Creating a Message Module Table 195
Assigning Values to Message Inserts. 196
Using Messages in Code. 197
Interpreting Run-Time Messages 198
Specifying National Language 199
Handling Message Output 199
Using LE/VSE MSGFILE 199
Using C Input/Output Functions. 200
Using COBOL Input/Output Statements . . . 201
Using PL/I Input/Output Statements 203
MSGFILE Considerations When Using PL/I . . 204
Examples Using Multiple Message Handling Callable Services 205
C Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG 205
COBOL Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and CEEMSG 207
PL/I Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG 210

Chapter 16. Using Date and Time Services 213
Understanding the Basics 213
Working with Date and Time Services 214
Date Limits 214
Picture Character Terms and Picture Strings . . 215
Notation for Eras 215
Performing Calculations on Date and Time Values 216
Century Window Routines 216
National Language Support for Date and Time Services 217
Examples Using Date and Time Callable Services 217
Examples Illustrating Calls to CEEQCEN and CEESCEN 219
Examples Illustrating Calls to CEESECS . . . 222
Examples Illustrating Calls to CEESECS and CEEDATM 226
Examples Illustrating Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM. 231
Example Illustrating Calls to CEEDAYS, CEEDATE, and CEEDYWK. 238

Chapter 17. National Language Support Services. 247
Understanding the Basics 247
Setting the National Language. 248
Setting the Country Code 248
Euro Support 249
Combining National Language Support and Date and Time Services 249
Calls to CEE5CTY, CEEFMDT, and CEEDATM in C 249
Calls to CEE5CTY, CEEFMDT, and CEEDATM in COBOL 252
Example Using CEE5CTY, CEEFMDT, and CEEDATM in PL/I 254

Chapter 18. Locale Callable Services 257
Understanding the Basics 257
Developing Internationalized Applications. . . . 258
Examples of Using Locale Callable Services . . . 258
Examples Illustrating Calls to CEEFMON . . . 258
Examples Illustrating Calls to CEEFTDS . . . 261
Examples Illustrating Calls to CEELCNV and CEESETL. 264
Examples Illustrating Calls to CEEQDTC and CEESETL. 267
Examples Illustrating Calls to CEESCOL . . . 270
Examples Illustrating Calls to CEESETL and CEEQRYL 273
Examples Illustrating Calls to CEEQRYL and CEESTXF. 276

Chapter 19. General Callable Services 281
CEE5DMP Callable Service 281
CEE5PRM Callable Service 282
CEE5PRML Callable Service 282
CEE5TSTG Callable Service. 282
CEE5USR Callable Service 282
CEEGPID Callable Service 282
CEERAN0 Callable Service 283
CEETEST Callable Service 283
Examples of Using Basic Callable Services. . . . 283

Chapter 20. Math Services 287
Understanding the Basics 287
Call Interface to Math Services 289
  Parameter Types: parm1 Type and parm2 Type 289
Examples of Calling Math Services 290
  Calling CEESSLOG in C. 290
  Calling CEESSLOG in COBOL. 291
  Calling CEESSLOG in PL/I. 292

Part 4. Using Interfaces to Other Products. 293

Chapter 21. Compatibility with Other Products 295
Required Licensed Programs 295
Optional Licensed Programs 295

Chapter 22. Running Applications under CICS 297
Understanding the Basics 297
  CICS Partition 297
  CICS Transaction 297
  CICS Run Unit 297
  Running LE/VSE Applications under CICS . . 298
Developing an Application under CICS. 298
  COBOL Coding Considerations under CICS . . 299
  PL/I Coding Considerations under CICS . . . 299
  Link-Edit Considerations under CICS 299
  Specifying Run-Time Options under CICS. . . 300
  Accessing DL/I Databases from CICS 302
  Using Callable Services under CICS 303
  DOS/VS COBOL Compatibility Considerations 303
  Using Math Services in PL/I under CICS . . . 303
  Coding Program Termination in PL/I under CICS 303
Storage Management 303
  CICS Short-on-Storage Condition. 303
  PL/I Storage Considerations under CICS . . . 304
Condition Handling under CICS 305
  PL/I Considerations for Using the CICS HANDLE ABEND Command 305
  Effect of the CICS HANDLE ABEND Command 306
  Effect of CICS HANDLE CONDITION and CICS HANDLE AID 306
  Restrictions on User-written Condition Handlers 306
  CICS Transaction Abend Codes 307
  Using the CBLPSHPOP Run-Time Option under CICS 307
  Restrictions on Assembler User Exits under CICS 307
  Ensuring Transaction Rollback under CICS . . 307
Run-Time Output under CICS. 308
  Message Handling under CICS 308
  Dump Services under CICS. 309
Support for Calls within the Same HLL under CICS 309
  C 309
  COBOL 309
  PL/I 310

Chapter 23. Running Applications with DB2 311
Understanding the Basics 311
  LE/VSE Support for DB2 Applications 311
  Specifying Run-Time Options with DB2 . . . 311
  Condition Handling under DB2 311

Chapter 24. Running Applications with DL/I 313
Understanding the Basics 313
Using the Interface between LE/VSE and DL/I . . 313
  CICS Considerations 313
  C Considerations 313
  PL/I Considerations 314
  Specifying Run-Time Options with DL/I . . . 314
Condition Handling with DL/I 314

Part 5. Specialized Programming Tasks 317

Chapter 25. Using Run-Time User Exits 319
Understanding the Basics 319
User Exits Supported under LE/VSE 319
  Using the Assembler User Exit CEEBXITA. . . 320
  Using the HLL Initialization Exit CEEBINT . . 320
  Using Sample Assembler User Exits 320
  CEEBINT High-Level Language User Exit Interface 332

Chapter 26. Assembler Considerations 335
Understanding the Basics 335
  Compatibility Considerations 335
Register Conventions. 335
Considerations for Coding or Running Assembler Routines 336
  Condition Handling 336
  Access to the Inbound Parameter String . . . 336
  Overlay Programs. 337
  CEESTART, CEEMAIN, and CEEFMAIN . . . 337
LE/VSE Library Routine Retention 337
  Using Library Routine Retention 338
  Library Routine Retention and Preinitialization 338
  CEELRR Macro— Initialize/Terminate LE/VSE Library Routine Retention 339
Assembler Macros. 341

CEEENTRY Macro— Generate an LE/VSE-Conforming Prolog 341
CEETERM Macro— Terminate an LE/VSE-Conforming Routine 343
CEECAA Macro— Generate a CAA Mapping 344
CEECIB Macro— Generate a CIB Mapping . . 345
CEEDSA Macro— Generate a DSA Mapping 345
CEEPPA Macro— Generate a PPA 345
CEELOAD Macro— Dynamically Load a Routine 348
CEEFETCH Macro— Dynamically Load a Routine that Can Be Later Deleted 350
CEERELES Macro— Dynamically Delete a Routine 353
Example of Assembler Main Routine 355
Example of an Assembler Main Calling an Assembler Subroutine 356
Invoking Callable Services from Assembler Routines 359
System Services Available to Assembler Routines 359

Chapter 27. Using Preinitialization Services 363

Understanding the Basics 363
Compatibility 364
Using Preinitialization 364
Using the PIPI Table 364
Reentrancy Considerations 366
User Exit Invocation 366
Stop Semantics 367
Specifying Run-Time Options and Program Arguments 367
CEEPIPI Interface 368
Initialization. 369
Application Invocation 373
Termination 379
Adding an Entry to the PIPI Table 380
Deleting an Entry from the PIPI Table 381
Service Routines 382
An Example Program Invocation of CEEPIPI . . . 387
HLLPIPI Examples 390

Chapter 28. Using Nested Enclaves 393

Understanding the Basics 393
COBOL Considerations 393
Determining the Behavior of Child Enclaves . . . 393
Creating Child Enclaves Using EXEC CICS LINK or EXEC CICS XCTL 394
Creating Child Enclaves Using the C system() Function 395
Other Nested Enclave Considerations 396
What the Enclave Returns from CEE5PRM . . 396
Finding the Return and Reason Code from the Enclave 397
Assembler User Exit 397
MSGFILE Considerations 397
AMODE Considerations 397

Part 6. Appendixes 399

Appendix A. Guidelines for Writing Callable Services. 401

Appendix B. Using Operating System and Subsystem Parameter List Formats. 403

C Parameter Passing Considerations. 403
C PLIST and EXECOPS Interactions 405
COBOL Parameter Passing Considerations . . . 406
PL/I Main Procedure Parameter Passing Considerations 406

Appendix C. Sort and Merge Considerations. 409

Understanding the Basics 409
Invoking DFSORT/VSE Directly 409
Using the COBOL SORT and MERGE Verbs . . . 409
User Exit Considerations 410
Condition Handling Considerations 410
Using the PL/I PLISRTx Interface 411
User Exit Considerations 411
Condition Handling Considerations 412
Storage Use during a Sort or Merge Operation . . 413
Sorting under CICS 413

Appendix D. LE/VSE Macros 415

Language Environment Glossary . . . 417

Index 427

Figures

1. Components of LE/VSE. 4
2. LE/VSE's Common Run-Time Environment 4
3. Example of Link-Editing a Fetchable Phase 8
4. Prelinker Map. 13
5. JCL for Prelinking a C Program 19
6. Accepting the Default Run-Time Options 21
7. Overriding the Default Run-Time Options 22
8. Overriding the Default Run-Time Options for COBOL 22
9. Basic Linkage Editor Processing. 23
10. Creating a Phase 25
11. Using the INCLUDE Linkage Editor Control Statement 26
12. Link-Edit Listing and Module Map 27
13. Link-Edit Map of a COBOL/VSE Program with Unresolved Weak External References . . 28
14. IBM-Supplied Batch Installation Default Options Source Program, CEEDOPT 41
15. IBM-Supplied CICS Installation Default Options Source Program, CEECOPT 42
16. IBM-Supplied Application Default Options Source Program, CEEUOPT 43
17. Sample of LE/CICS-Wide Options Printed to Console 46
18. Call Terminology Refresher 50
19. Argument-Passing Styles in LE/VSE 51
20. LE/VSE ILC—Only One Run-Time Environment to Initialize 66
21. Program Management Model Illustration of Resource Ownership 76
22. Overview of the Full Language Environment Program Management Model 80
23. LE/VSE Stack Storage Model 84
24. LE/VSE Heap Storage Model 86
25. C Example Using CEEGTST and CEEFRST to Build a Linked List 90
26. COBOL Example Using CEEGTST and CEEFRST to Build a Linked List 92
27. PL/I Example Using CEEGTST and CEEFRST to Build a Linked List 94
28. C Example Illustrating Calls to CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST, and CEEDSHP 96
29. COBOL Example Illustrating Calls to CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST and CEEDSHP 98
30. PL/I Example Illustrating Calls to CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST and CEEDSHP 100
31. Condition Processing 109
32. Queues of User-Written Condition Handlers 115
33. Scenario 1: Division by Zero with No User Condition Handlers Present. 117
34. Scenario 2: Division by Zero with a User-Written Condition Handler Present in Routine A. 118
35. Scenario 3: Division by Zero with a User Handler Present in Routine B 119
36. C370A Routine 125
37. C370B Routine 126
38. C370C Routine 126
39. C Condition Handling Example 127
40. COBOLA Program 130
41. COBOLB Program 130
42. COBOLC Program 131
43. PL/I Condition Processing 133
44. PL/I Condition Handling Example 135
45. Restricted type_of_move If COBOL Nested Programs Are Present. 141
46. Handle and Resume Cursor Movement as a Condition Is Handled. 142
47. EXCOND Routine (C). 143
48. EXCOND Routine (COBOL) 145
49. DIVZERO Routine (COBOL) 146
50. USRHDLR Routine (COBOL) 147
51. C Example of a main() Routine That Calls a Function and Registers a Condition Handler for an Out-of-Storage Condition 150
52. C User-Written Condition Handler Registered for the Out-of-Storage Condition 152
53. COBOL Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler. 153
54. COBOL User-Written Condition Handler Registered for the Out-of-Storage Condition . 155
55. COBOL Subroutine that Causes Out-of-Storage Condition 157
56. Sample C Calls to CEEHDLR, CEESGL, CEEGQDT, and CEEMRCR 158
57. COBOL Example of a Main Routine that Registers User-Written Condition Handler and Causes Divide-by-Zero Condition . . . 161
58. Assembler Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler. 165
59. LE/VSE Condition Token 172
60. C Example Testing for CEEGTST Symbolic Feedback Code CEE0P3 178
61. COBOL Example Testing for CEESDEXP Symbolic Feedback Code CEE1UR 179
62. Wrong Placement of COBOL COPY Statements for Testing Feedback Code . . . 180
63. PL/I Example Testing for Symbolic Feedback Code CEE000 181
64. Structure of Abend Qualifying Data 183
65. Example of a Message Source File 186
66. Example of a Message Module Table with One Language 195
67. Example of a Message Module Table with Two Languages 195
68. Assembling and Link-Editing a Message Phase Table 196

© Copyright IBM Corp. 1991, 2005

69. Example of Assigning Values to Message Inserts 197
70. Example of Link-Editing and Running the TEST PL/I Program 197
71. C Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG 205
72. COBOL Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and CEEMSG 207
73. PL/I Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG 210
74. Performing Calculations on Dates. 216
75. C Example of Querying and Changing the Century Window 219
76. COBOL Example of Querying and Changing the Century Window 220
77. PL/I Example of Querying and Changing the Century Window 221
78. Calls to CEESECS in C 222
79. Calls to CEESECS in COBOL 223
80. Calls to CEESECS in PL/I 225
81. Calls to CEESECS and CEEDATM in C 226
82. Calls to CEESECS and CEEDATM in COBOL 228
83. Calls to CEESECS and CEEDATM in PL/I 230
84. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in C. 231
85. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in COBOL 233
86. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in PL/I 236
87. Calls to CEEDAYS, CEEDATE, and CEEDYWK in C 238
88. Calls to CEEDAYS, CEEDATE, and CEEDYWK in COBOL 240
89. Calls to CEEDAYS, CEEDATE, and CEEDYWK in PL/I 243
90. Calls to CEECBLDY in COBOL 245
91. Querying and Setting the Country Code and Getting the Date and Time Format in C . . . 250
92. Querying and Setting the Country Code and Getting the Date and Time Format in COBOL. 252
93. Querying and Setting the Country Code and Getting the Date and Time Format in PL/I. . 254
94. Calls to CEEFMON in COBOL. 259
95. Calls to CEEFMON in PL/I. 260
96. Calls to CEEFTDS in COBOL 261
97. Calls to CEEFTDS in PL/I 263
98. Calls to CEELCNV and CEESETL in COBOL 264
99. Calls to CEELCNV and CEESETL in PL/I 266
100. Calls to CEEQDTC and CEESETL in COBOL 267
101. Calls to CEEQTDC and CEESETL in PL/I 269
102. Calls to CEESCOL in COBOL 270
103. Calls to CEESCOL in PL/I 272
104. Calls to CEESETL and CEEQRYL in COBOL 273
105. Calls to CEESETL and CEEQRYL in PL/I 275
106. Calls to CEEQRYL and CEESTXF in COBOL 276
107. Calls to CEEQRYL and CEESTXF in PL/I 278
108. C Routine with a Call to CEEFMDT 284
109. COBOL Program with a Call to CEEFMDT 285
110. PL/I Routine with a Call to CEEFMDT 286
111. C Call to CEESSLOG—Logarithm Base e 290
112. Call to CEESSLOG—Logarithm Base e in COBOL 291
113. Call to CEESSLOG—Logarithm Base e in PL/I 292
114. Format of Messages Sent to CESE. 308
115. Location of User Exits. 321
116. Interface for CEEBXITA Assembler User Exit 324
117. CEEAUE_FLAGS Format 325
118. Exit_list and Hook_exit Control Blocks 333
119. Example of Simple Main Assembler Routine 355
120. Example of an Assembler Main Routine Calling a Subroutine 356
121. Example of a Called Assembler Subroutine 357
122. Sample Invocation of a Callable Service from Assembler 359
123. Format of Service Routine Vector 382
124. Assembler Driver that Creates a Preinitialized Environment. 387
125. C Subroutine Called by ASMPIPI 390
126. COBOL Program Called by ASMPIPI 390
127. PL/I Routine Called by ASMPIPI 391
128. Some C Parameter Passing Styles 403
129. Accessing Parameters Using Macros __R1 and __osplist 404
130. Examples of Casting and Dereferencing 405
131. Format of Sort Parameter List under CICS 414

Tables

1. LE/VSE-Conforming Languages xvi
2. LE/VSE Publications xxi
3. z/VSE Publications. xxi
4. IBM C for VSE/ESA Publications xxi
5. IBM COBOL for VSE/ESA Publications xxii
6. IBM PL/I for VSE/ESA Publications xxii
7. Debug Tool for VSE/ESA Publications xxii
8. Link-Edit Default Entry Point by Language 6
9. Prelinker Options 19
10. Files Used for Link-Editing 23
11. Linkage Editor Control Statements 24
12. Summary of LE/VSE Run-Time Options 33
13. Formats for Specifying Run-Time Options and Program Arguments 38
14. Semantic Terms and Methods for Passing Arguments in LE/VSE 50
15. Default Passing Style per HLL 51
16. Coding a Main Routine to Receive an Inbound Parameter List in Batch without DL/I. 52
17. Coding a Main Routine to Receive an Inbound Parameter List in Batch with DL/I 53
18. Coding a Main Routine to Receive an Inbound Parameter List in CICS. 54
19. Coding a Main Routine to Receive an Inbound Parameter List in Batch 54
20. Summary of Enclave Reason Codes 70
21. Termination Behavior for Unhandled Conditions of Severity 2 or Greater 71
22. Abend Codes Used by LE/VSE when the Assembler User Exit Requests an Abend . . . 71
23. Abend Code Values Used by LE/VSE with ABTERMENC(ABEND) 72
24. Program Interrupt Codes in a Non-CICS Environment 73
25. Usage of Stack and Heap Storage by LE/VSE-Conforming Languages 81
26. Heap IDs Recognized by LE/VSE Heap Manager 86
27. COBOL Storage Usage 88
28. LE/VSE Default Responses to Unhandled Conditions 111
29. T_I_U Condition Representation 112
30. T_I_S Condition Representation 113
31. C Conditions and Default System Actions 122
32. Mapping of S/370 Exceptions to C Signals 128
33. Mapping of Abend Signals to C Signals 128
34. Symbolic Feedback Codes Associated with CEEGTST 175
35. LE/VSE Condition Tokens and C Signals 182
36. LE/VSE Run-Time Message Severity Codes 198
37. Message File Default Attributes 199
38. Defining an I/O Device for the Message File 200
39. C Message Output 201
40. C Redirected Stream Output 201
41. Run-time Message and DISPLAY Destinations for OUTDD and MSGFILE filename Specifications 202
42. LE/VSE Locale Callable Services and Equivalent C Library Routines 257
43. Required Licensed Programs for LE/VSE 295
44. Optional Licensed Compiler Programs for LE/VSE 295
45. Other Licensed Programs for LE/VSE 295
46. Run-Time Option Behavior under CICS 301
47. User Exits Supported under LE/VSE 319
48. Sample Assembler User Exits for LE/VSE 320
49. Parameter Values in the Assembler User Exit (Part 1) 329
50. Parameter Values in the Assembler User Exit (Part 2) 331
51. LE/VSE's Equivalent Host Services 359
52. Invocation of User Exits during Process and Enclave Initialization and Termination . . . 366
53. Preinitialization Services Accessed Using CEEPIPI 368
54. Return and Reason Codes 383
55. Return and Reason Codes 384
56. Return and Reason Codes 385
57. Return and Reason Codes 386
58. Return and Reason Codes 386
59. Unhandled Condition Behavior in a system()-Created Child Enclave 395
60. Determining the Command-Line Equivalent 396
61. Determining the Order of Run-Time Options and Program Arguments. 397
62. Interactions of C PLIST and EXECOPS (`#pragma runopts`) 406
63. Interactions of SYSTEM and NOEXECOPS 407
64. DFSORT/VSE Exit Called as a Function of a PLISRTx Interface Call 412

© Copyright IBM Corp. 1991, 2005

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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 of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, U.S.A.

Any pointers in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement. IBM accepts no responsibility for the content or use of non-IBM Web sites specifically mentioned in this publication or accessed through an IBM Web site that is mentioned in this publication.

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 Deutschland Informationssysteme GmbH
Department 0215
Pascalstr. 100
70569 Stuttgart
Germany
```

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

Programming Interface Information

This book is intended to help with application programming. This book documents General-Use Programming Interface and Associated Guidance Information provided by IBM Language Environment for VSE/ESA.

General-Use programming interfaces allow the customer to write programs that obtain the services of IBM Language Environment for VSE/ESA.

Trademarks and Service Marks

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

AT	Integrated Language Environment	VSE/ESA
C/370	Language Environment	z/OS
CICS	OS/390	zSeries
CICS/VSE	OS/400	z/VM
DB2	SAA	z/VSE
DFSORT	System/370	
IBM	VisualAge	

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, and/or other countries.

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

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

Other company, product, and service names, may be trademarks or service marks of others.

About This Book

z/VSE is the successor to IBM's VSE/ESA product. Many products and functions supported on z/VSE may continue to use VSE/ESA in their names.

z/VSE can execute in 31-bit mode only. It does not implement z/Architecture, and specifically does not implement 64-bit mode capabilities.

z/VSE is designed to exploit select features of IBM eServer zSeries hardware.

This book contains information about linking, running, and using services within the IBM Language Environment for VSE/ESA (LE/VSE) environment, the LE/VSE program management model, and language- and operating system-specific information where applicable. For application programming, you will need to use both this book and *LE/VSE Programming Reference*. *LE/VSE Programming Reference* contains more detailed information, as well as specific syntax for using run-time options and callable services. *LE/VSE Writing Interlanguage Communication Applications* provides information to help you create and run interlanguage communication (ILC) applications.

What Is LE/VSE?

LE/VSE is a set of common services and language-specific routines that provide a single run-time environment for applications written in *LE/VSE-conforming* versions of the C, COBOL, and PL/I high level languages (HLLs), and for many applications written in previous versions of COBOL. (For a list of LE/VSE-conforming languages, and a description of compatibility with previous versions of COBOL, see "LE/VSE-Conforming Languages" on page xvi.) LE/VSE also supports applications written in assembler language using LE/VSE-provided macros and assembled using High Level Assembler (HLASM).

Prior to LE/VSE, each programming language provided its own separate run-time environment. LE/VSE combines essential and commonly-used run-time services—such as message handling, condition handling, storage management, date and time services, and math functions—and makes them available through a set of interfaces that are consistent across programming languages. With LE/VSE, you can use one run-time environment for your applications, regardless of the application's programming language or system resource needs, because most system dependencies have been removed.

Services that work with only one language are available within language-specific portions of LE/VSE.

LE/VSE consists of:

- Basic routines for starting and stopping programs, allocating storage, communicating with programs written in different languages, and indicating and handling error conditions.
- Common library services, such as math services and date and time services, that are commonly needed by programs running on the system. These functions are supported through a library of callable services.

© Copyright IBM Corp. 1991, 2005

- Language-specific portions of the common run-time library.

LE/VSE is the implementation of Language Environment on the VSE platform. Language Environment is also offered on platforms z/OS and VM, and on OS/400 as Integrated Language Environment.

LE/VSE-Conforming Languages

An LE/VSE-conforming language is any HLL that adheres to the LE/VSE common interface. Table 1 lists the LE/VSE-conforming language compiler products you can use to generate applications that run with LE/VSE Release 4.

Table 1. LE/VSE-Conforming Languages

Language	LE/VSE-Conforming Language	Minimum Release
C	IBM C for VSE/ESA	Release 1
COBOL	IBM COBOL for VSE/ESA	Release 1
PL/I	IBM PL/I for VSE/ESA	Release 1

Any HLL not listed in Table 1 is known as a *non-LE/VSE-conforming* or, alternatively, a *pre-LE/VSE-conforming* language. Some examples of non-LE/VSE-conforming languages are:
- C/370
- DOS/VS COBOL
- VS COBOL II
- DOS PL/I
- DOS/VS RPG II

Only the following products can generate applications that run with LE/VSE:
- LE/VSE-conforming languages
- HLASM using LE/VSE-provided macros (for details, see *LE/VSE Programming Guide*)
- DOS/VS COBOL and VS COBOL II, with some restrictions (see LE/VSE Compatibility with Previous Versions of COBOL below).

LE/VSE Compatibility with Previous Versions of COBOL

Although DOS/VS COBOL and VS COBOL II are non-LE/VSE-conforming languages, many applications generated with these compilers can run with LE/VSE without recompiling. For details about compatibility, see *LE/VSE Run-Time Migration Guide*.

However relinking under LE/VSE is the *minimum* effort in order to migrate run-time, and involve LE/VSE COBOL-compatibility routines (rather than the old and unsupported library routines of non-LE/VSE conforming COBOL compilers). This particularily applies to NORES-compiled units or applications that involve former initialization techniques such as ILBDSET0. There are even restrictions with this approach, such as:
- No use of 4-digit dates.
- No exploitation of LE/VSE functionality.
- Interlanguage communication capabilities, and so on.

Therefore you are *strongly recommended* to carry out a (subsequent) full migration to a higher ANSI standard and LE/VSE-conforming COBOL compiler (COBOL for VSE/ESA).

VS COBOL II can also dynamically call some LE/VSE date and time callable services. For details, see *LE/VSE Programming Reference*.

Terms Used in This Book

Unless otherwise stated, the following terms are used in this book to refer to the specified languages:

Term...	**Refers to the language supported by...**
C	The IBM C for VSE/ESA compiler
COBOL	The IBM COBOL for VSE/ESA and VS COBOL II compilers
PL/I	The IBM PL/I for VSE/ESA compilers

For a list of LE/VSE-conforming language compilers, see "LE/VSE-Conforming Languages" on page xvi.

How This Book Is Organized

This book is organized as follows:

- Part 1 is a basic introduction to LE/VSE and discusses prelinking, linking, and running with LE/VSE, as well as using LE/VSE run-time options.
- Part 2 describes how to prepare an application to run in LE/VSE.
- Part 3 describes LE/VSE concepts, services, and models, including initialization and termination, program management model, storage, condition handling, messages, callable services, and math services.
- Part 4 explains using interfaces to other products such as CICS, DB2, and DL/I.
- Part 5 addresses specialized programming tasks, such as using run-time user exits, assembler considerations, preinitialization services, and using nested enclaves.
- The various appendixes discuss writing callable services, using parameter list formats, sort and merge considerations, and LE/VSE macros.

How to Read the Syntax Diagrams

The following rules apply to the notation used in the syntax diagrams contained in this book:

- Read the syntax diagrams from left to right, top to bottom following the path of the line.
- Each syntax diagram begins with a double arrowhead (►►).
- An arrow (─►) at the end of a line indicates that the option, service, or macro syntax continues on the next line. A continuation line begins with an arrow (►─).
- If a syntax diagram contains too many items or groups to fit in the diagram, the syntax is shown by a main syntax diagram and one or more syntax fragments. A syntax fragment is referred to in the main diagram by its fragment name between two vertical bars (|).

  Each syntax fragment appears below the main syntax diagram, and begins and ends with a vertical bar (|). A heading above the fragment indicates the name of the fragment.

  Read each syntax fragment as though it were imbedded in the main syntax diagram.
- IBM-supplied default keywords appear **above** the main path or options path (see the sample on page xviii). In the parameter list, IBM-supplied default choices are underlined.
- Keywords appear in nonitalic capital letters and should be entered exactly as shown. However, some keywords may be abbreviated by truncation from the right as long as the result is unambiguous. In this case, the unambiguous truncation is shown in capital letters in the keyword, for example:

  ```
  ANyheap
  ```
- Words in lowercase letters represent user-defined parameters or suboptions.

- Enter parentheses, arithmetic symbols, colons, semicolons, commas, and greater-than signs where shown.
- Required parameters appear on the same horizontal line (the main path) as the option, service, or macro:

```
►►──OPTION──required_parameter─────────────────────────────────────────────►◄
```

- If you can choose from two or more parameters, the choices are stacked one above the other.

  If choosing one of the items is optional, the entire stack appears below the main line.

```
►►──OPTION──┬─────────────────────┬────────────────────────────────────────►◄
            ├─optional_parameter_1─┤
            ├─optional_parameter_2─┤
            └─optional_parameter_3─┘
```

  If you *must* choose one of the items, one item of the stack appears on the main path:

```
►►──OPTION──┬─required_choice_1─┬──────────────────────────────────────────►◄
            ├─required_choice_2─┤
            └─required_choice_3─┘
```

- An arrow returning to the left above a line indicates that an item can be repeated:

```
            ┌─────────────────┐
            ▼                 │
►►──OPTION────repeatable_item─┴────────────────────────────────────────────►◄
```

  OR

```
►►──OPTION──┬───────────────────────┬──────────────────────────────────────►◄
            │ ┌─────────────────┐   │
            │ ▼                 │   │
            └───repeatable_item─┴───┘
```

- A comma or semicolon included in the repeat symbol indicates a separator that you must include between repeated parameters. These separators must be coded where shown.
- When entering commands, parameters and keywords must be separated by at least one blank if there is no intervening punctuation.
- A double arrow (─►◄) at the end of a line indicates the end of the syntax diagram.

The following example demonstrates how to read the syntax notation. Numbers in the example correspond to explanations supplied below the example.

```
                                                                 (6)
           (1)   (2)                (4)            ┌─ANYWHERE─┐     ┌─FREE───┐
►►──ANyheap────(──┬───────────────┬─,─┬──────────┬─,─┼──────────┼─,─┼────────┼───►
                  │            (3)│   └─incr_size┘   ├─ANY──────┤   │   (5)  │
                  └─init_size─────┘                  └─BELOW────┘   └─KEEP───┘

►─)────────────────────────────────────────────────────────────────────────►◄
```

Notes:

1 Keyword with minimum unambiguous truncation shown in capital letters

2 Opening parenthesis (must be specified if any parameters are specified)

3 Optional parameter

4 Comma (must be specified if there are parameters that follow)

5 Optional keyword

6 Optional keyword (IBM-supplied default)

Where to Find More Information

These are the manuals that describe LE/VSE:

Table 2. LE/VSE Publications

Publication	**Form Number**
LE/VSE Fact Sheet	GC33-6679
LE/VSE Concepts Guide	GC33-6680
LE/VSE Customization Guide	SC33-6682
LE/VSE Programming Guide	SC33-6684
LE/VSE Programming Reference	SC33-6685
LE/VSE C Run-Time Programming Guide	SC33-6688
LE/VSE C Run-Time Library Reference	SC33-6689
LE/VSE Debugging Guide and Run-Time Messages	SC33-6681
LE/VSE Writing Interlanguage Communication Applications	SC33-6686
LE/VSE Run-Time Migration Guide	SC33-6687
LE/VSE Licensed Program Specifications	GC33-6683

These are the z/VSE manuals to which you might need to refer:

Table 3. z/VSE Publications

Publication	**Form Number**
z/VSE Administration	SC33-8224
z/VSE Messages and Codes, Volume 1	SC33-8226
z/VSE Messages and Codes, Volume 2	SC33-8227
z/VSE Messages and Codes, Volume 3	SC33-8228
z/VSE Planning	SC33-8221
z/VSE System Control Statements	SC33-8225
z/VSE System Macros Reference	SC33-8230
z/VSE System Macros User's Guide	SC33-8236
z/VSE System Upgrade and Service	SC33-8223
VSE/VSAM User's Guide and Application Programming	SC33-8246
VSE/VSAM Commands	SC33-8245
TCP/IP for VSE/ESA IBM Program Setup and Supplementary Information	SC33-6601

These are the manuals that describe IBM C for VSE/ESA:

Table 4. IBM C for VSE/ESA Publications

Publication	**Form Number**
Licensed Program Specifications	GC09-2421
Installation and Customization Guide	GC09-2422
Migration Guide	SC09-2423

© Copyright IBM Corp. 1991, 2005

Table 4. IBM C for VSE/ESA Publications (continued)

Publication	Form Number
User's Guide	SC09-2424
Language Reference	SC09-2425
Diagnosis Guide	GC09-2426

These are the manuals that describe IBM COBOL for VSE/ESA:

Table 5. IBM COBOL for VSE/ESA Publications

Publication	Form Number
General Information	GC33-6679
Licensed Program Specifications	GC33-6680
Migration Guide	SC33-6682
Installation and Customization Guide	GC33-6680
Programming Guide	SC33-6684
Language Reference	SC33-6685
Diagnosis Guide	SC33-6684
Reference Summary	SX26-3834

These are the manuals that describe IBM PL/I for VSE/ESA:

Table 6. IBM PL/I for VSE/ESA Publications

Publication	Form Number
Fact Sheet	GC26-8052
Programming Guide	SC26-8053
Language Reference	SC26-8054
Licensed Program Specifications	GC26-8055
Migration Guide	SC33-6684
Installation and Customization Guide	SC26-8057
Diagnosis Guide	SC26-8058
Compile-Time Messages and Codes	SC26-8059
Reference Summary	SX26-3836

These are the manuals that describe Debug Tool for VSE/ESA:

Table 7. Debug Tool for VSE/ESA Publications

Publication	Form Number
User's Guide and Reference	SC26-8797
Installation and Customization Guide	SC26-8798
Fact Sheet	GC26-8925

You might also refer to the ...

z/VSE Home Page

z/VSE has a home page on the World Wide Web, which offers up-to-date information about VSE-related products and services, new z/VSE functions, and other items of interest to VSE users.

You can find the z/VSE home page at:

`http://www.ibm.com/servers/eserver/zseries/zvse/`

Softcopy Publications

The following collection kit contains the LE/VSE and LE/VSE-conforming language product publications:

VSE Collection, SK2T-0060

Summary of Changes

This section describes the changes introduced with the current and previous two editions of this manual.

Changes Introduced with Fifth Edition (March 2005)

These are the most important changes introduced with the fifth edition of this manual (covering LE/VSE 1.4.4):

- The name **VSE/ESA** has now changed to **z/VSE**. However, the names of many features and programs related to z/VSE remain unchanged (such as IBM Language Environment for VSE/ESA, IBM COBOL for VSE/ESA, Debug Tool for VSE/ESA, or CICS Transaction Server for VSE/ESA).
- The existing facility that utilizes VSE/POWER LSTQ to store LE/VSE dumps (tailored using the sub-options of the TERMTHDACT run-time option), is also available for LE/VSE *batch* processing. It can therefore be used optionally for LE/VSE CICS and batch environments.
- The figures showing CEEDOPT (the IBM-supplied batch installation default options source program), and CEECOPT (the IBM-supplied CICS installation default options source program), have been updated to show the use of the CEELOPT macro. For details, see Figure 14 on page 41 and Figure 15 on page 42.
- A description of the LE/VSE support for the Euro has been introduced. For details, see "Euro Support" on page 249.
- Callable service CEE5PRM has been changed back to its previous functionality, so that it can be used with a buffer of length 80 characters. For details, see "CEE5PRM Callable Service" on page 282.
- Callable service CEE5PRML is new. You can use it to clear a buffer of length 300 characters. For details, see "CEE5PRML Callable Service" on page 282.
- Callable service CEE5TSTG is new. You can use it to test the access that is available to a specified storage address. For details, see "CEE5TSTG Callable Service" on page 282.
- Assembler macro support via CEEFETCH and CEERELES now allows reentrant COBOL routines to be dynamically loaded and deleted. Before z/VSE 3.1, only PL/I and C FETCHABLE reentrant routines were supported. For details, see "CEEFETCH Macro— Dynamically Load a Routine that Can Be Later Deleted" on page 350 and "CEERELES Macro— Dynamically Delete a Routine" on page 353.

Changes Introduced with Fourth Edition (March 2003)

These are the most important changes introduced with the fourth edition (covering LE/VSE 1.4.3):

- For the CEE5PRM callable service, support is introduced for JCL invocation strings that are up to 300 characters in length (previously this was only 80 characters). For details, see *page 282*.
- Assembler macro CEECIB can be used to generate a condition information block (CIB) mapping. For details see *page 345*.

© Copyright IBM Corp. 1991, 2005

- Assembler macro CEEFETCH allows an LE/VSE-conforming assembler routine to dynamically load another LE/VSE-conforming HLL subroutine. This loaded LE/VSE-conforming HLL subroutine can then be later deleted using CEERELES. For details of:
  – CEEFETCH see *page 350*.
  – CEERELES see *page 353*.
- The currently-active version of LE/VSE is checked when the system is initialized using the CEEPIPI interface. As a result, a new Return Code 12 ("The version of the CEEXPIT macro used at assembly time is not supported by the version of LE/VSE that is currently running") has been added for the CEEPIPI (init_main), CEEPIPI (init_sub), and CEEPIPI (init_sub_dp) functions. For details, see *pages 369*, *370*, and *371* respectively.
- Calling CEEPIPI with a delete_entry request allows an entry to be deleted from the PIPI table. The entry is then available for subsequent CEEPIPI(add_entry) functions. For details, see *page 381*.

Changes Introduced with Third Edition (December 2001)

These were the most important changes introduced with the third edition (covering LE/VSE 1.4.1 and 1.4.2):

- LE/VSE's condition-handling facilities were enhanced by the addition of new callable services. For details, see *page 104.*
- The CEEENTRY macro was enhanced to allow:
  – The use of multiple base registers and RMODE specification.
  – Mulitple execution of CEEENTRY within a single assembly.

  For details, see *page 341.*
- Descriptions of the NAME and NAMEADDR options for the CEELOAD macro were changed. For details, see *page 348.*

Part 1. Linking and Running Applications with LE/VSE

Chapter 1. Introduction to LE/VSE 3

Chapter 2. Preparing to Link and Run under LE/VSE 5
Understanding the Basics 5
Planning to Link and Run 5
Link-Editing Single-Language Applications . . . 6
Link-Editing ILC Applications 6
Checking Which Run-Time Options Are in Effect . . 6
COBOL Considerations 7
COBOL Linking Considerations 7
PL/I Linking Considerations 7
Link-Editing Fetchable Phases 7
Fetching Phases with Different AMODEs 8
C AMODE/RMODE Considerations 9

Chapter 3. Prelinking an Application 11
Which Programs Need to Be Prelinked 11
What the Prelinker Does 11
Prelinking Process 11
Using the Prelinker Automatic Library Call. . . 12
LE/VSE Prelinker Map 13
Control Statement Processing 15
INCLUDE Control Statement 15
LIBRARY Control Statement 16
RENAME Control Statement. 16
Usage Notes 17
Mapping L-Names to S-Names 18
Running the Prelinker 19
Prelinker Options 19

Chapter 4. Linking and Running 21
Basic Linking and Running 21
Linking and Running an Existing Object Module 21
Overriding the Default Run-Time Options . . . 21
Providing Input to the Linkage Editor 22
Writing JCL for the Linkage Editor 23
Specifying the Files Used by the Linkage Editor 23
Specifying Linkage Editor Control Statements 24
Using the EXEC Statement 25
Using the PARM Parameter for the Linkage Editor 25
Example of Linkage Editor JCL. 25
Link-Editing Multiple Object Modules 26
Using the INCLUDE Statement. 26
Linkage Editor Module Map. 26
Detecting Link-Edit Errors 28
Running an Application 29
Specifying the Search Order 29
Specifying Run-Time Options 30
Specifying Run-Time Options in the EXEC Statement 30
Using the iconv Utility for C 31
Using the genxlt Utility for C 31

Chapter 5. Using Run-Time Options 33
Understanding the Basics. 33
Specifying Run-time Options 35
Order of Precedence 37
Specifying Suboptions in Run-Time Options . . 38
Specifying Run-Time Options and Program Arguments 38
C Compatibility Considerations. 39
COBOL Compatibility Considerations 39
PL/I Compatibility Considerations 39
CEEXOPT Invocation Syntax 40
Notes on CEEXOPT Invocation. 44
Performance Considerations 45
Printing CICS-Wide Run-Time Options to Console. 45

This section explains how to prelink, link, and run applications in LE/VSE. Prelinking, linking, and running commands, as well as an overview of run-time options, are included.

If you have an application that contains interlanguage calls, you might need to relink it to take advantage of the improved LE/VSE ILC support. See *LE/VSE Writing Interlanguage Communication Applications* for more information.

© Copyright IBM Corp. 1991, 2005

Chapter 1. Introduction to LE/VSE

LE/VSE provides a single *run-time environment* [1] for applications written in LE/VSE-conforming versions of the C, COBOL, and PL/I HLLs, and for many applications written in previous versions of COBOL. (For a list of LE/VSE-conforming languages, and a description of compatibility with previous versions of COBOL, see “LE/VSE-Conforming Languages” on page xvi.) LE/VSE also supports applications written in assembler language using LE/VSE-provided macros and assembled using HLASM.

Note!

The IBM DOS/VS RPG II programming language is *not* a Language Environment-conforming language. You can therefore *not* use RPG II directly with other Language Environment-conforming languages that support ILC calls.

Prior to LE/VSE, each of the HLLs had to provide a separate run-time environment. With LE/VSE, routines call one another within one common run-time environment, regardless of the LE/VSE-conforming HLL they are written in. Routines follow common calling conventions that standardize the way routines call one another and make interlanguage communication (ILC) in mixed-language applications easier, more efficient, and more consistent.

LE/VSE also combines essential and commonly used run-time services, such as routines for run-time message handling, condition handling, storage management, date and time services, and math functions, and makes them available through a set of interfaces that are consistent across programming languages. With LE/VSE, you can use one run-time environment for your applications, regardless of the application’s programming language or system resource needs because most system dependencies have been removed.

Services that work with only one language are also available within language-specific portions of LE/VSE.

LE/VSE consists of:

- Basic routines that support starting and stopping programs, allocating storage, communicating with programs written in different languages, and indicating and handling error conditions.
- Common library services, such as math services and date and time services, that are commonly needed by programs running on the system. These functions are supported through a library of callable services.
- Language-specific portions of the common run-time library.

Figure 1 on page 4 shows the separate components that make up LE/VSE.

1. Terms that might be unfamiliar to you are highlighted (like *run-time environment*) the first time they appear. These terms are defined in the “Language Environment Glossary” on page 417.

© Copyright IBM Corp. 1991, 2005

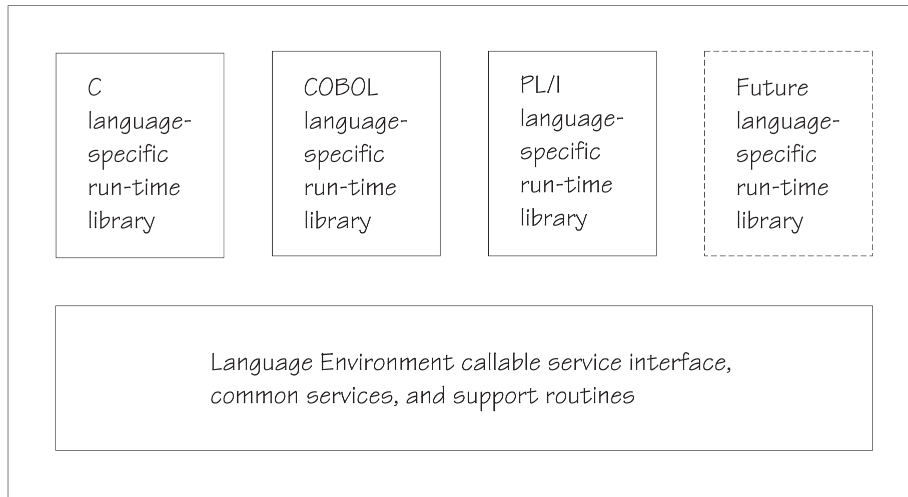


Figure 1. Components of LE/VSE

Figure 2 illustrates the common environment that LE/VSE creates.

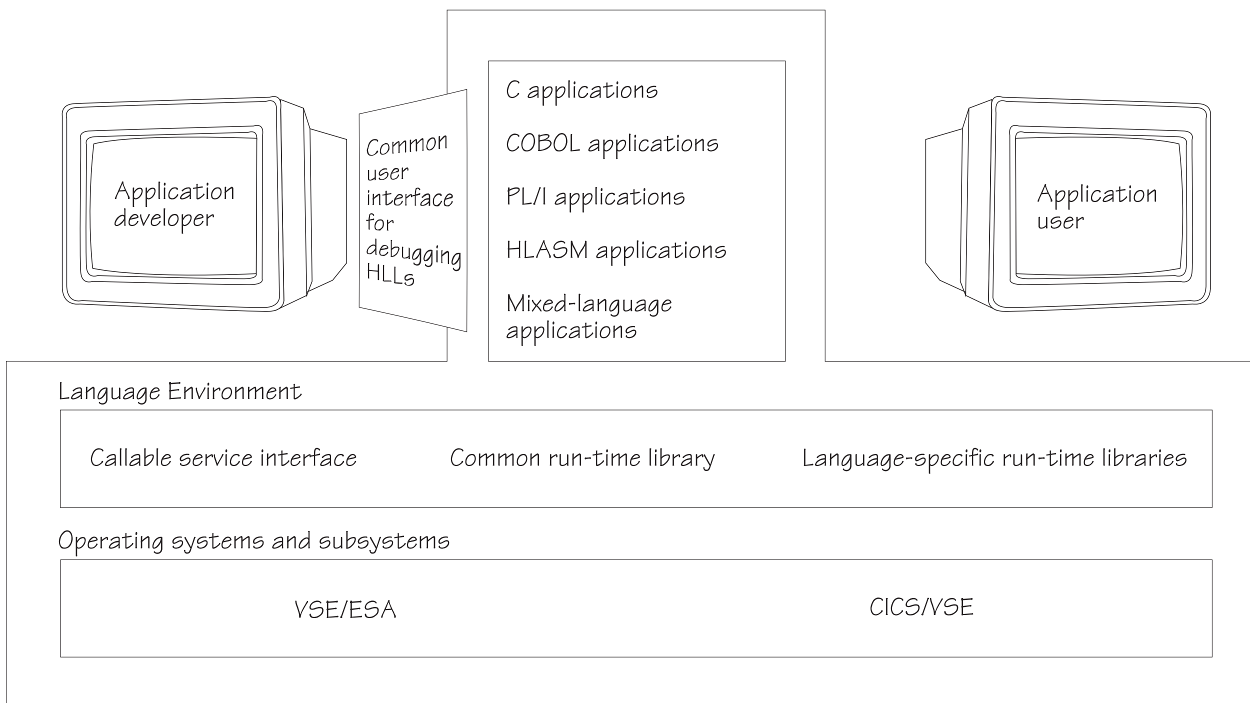


Figure 2. LE/VSE's Common Run-Time Environment

For a complete list of operating systems and subsystems supported by LE/VSE, see Chapter 21, "Compatibility with Other Products," on page 295.

Chapter 2. Preparing to Link and Run under LE/VSE

This chapter discusses what you need to know before linking and running applications under LE/VSE. After LE/VSE is installed on your system, you should run an existing application in the LE/VSE run-time environment. Although you need to link to different libraries under LE/VSE, the procedure is similar to that used in pre-LE/VSE versions of C, COBOL, and PL/I. To help you get started, this chapter describes the following common initial link and run tasks, which you might want to try before reading further:

- Linking and running an existing object module and accepting the default LE/VSE run-time options
- Linking and running an existing object module and specifying new LE/VSE run-time options
- Calling an LE/VSE service

These are basic tasks intended to help give you an idea of what running an application under LE/VSE is like. They are not intended to illustrate every aspect of linking and running you might want to learn. Detailed instructions about linking and running existing and new applications are provided in Chapter 4, "Linking and Running," on page 21 and *z/VSE System Control Statements*

Understanding the Basics

LE/VSE library routines are divided into two categories: *resident routines* and *dynamic routines*. The resident routines are linked with the application and include such things as initialization/termination routines and pointers to callable services. The dynamic routines are not part of the application and are dynamically loaded during run time. For more information, see *LE/VSE Customization Guide*.

The way LE/VSE code is packaged keeps the size of application phases small. When maintaining dynamic library code, you need not relink-edit the application code except under special circumstances, such as when you use an earlier version of code.

The linkage editor converts an object module into an executable phase and stores it in a VSE Librarian sublibrary. The phase becomes a permanent member of the sublibrary and can be run at any time, either in the job that created it, or in any other job.

Input to the linkage editor consists of object modules and control statements that specify how the input is to be processed. The output from the linkage editor can be a single phase or multiple phases (using the PHASE linkage editor control statement).

Planning to Link and Run

Before linking and running applications, you need to determine the name of the sublibrary or sublibraries where LE/VSE routines are located. If LE/VSE has been installed in the default sublibraries, all resident LE/VSE routines and dynamic LE/VSE routines are located in the PRD2.SCEEBASE sublibrary.

© Copyright IBM Corp. 1991, 2005

Link-Editing Single-Language Applications

The entry point for an application is determined in one of two ways:
- The order of the routines presented to the linkage editor
- Explicit specification of the entry point by a linkage editor control statement

Table 8 identifies the default entry points for LE/VSE-conforming HLLs. The main routine entry point in an application is nominated by the END text record in the object deck produced by the compiler.

Table 8. Link-Edit Default Entry Point by Language

Language	Default Entry Point
C	CEESTART
COBOL	The name of the first object module presented to the linkage editor
PL/I	CEESTART

A copy of CEESTART resides in the LE/VSE library. Do not explicitly include it in a link-edit of your object modules, even for LE/VSE-enabled languages. The compilers generate CEESTART when necessary.

You must link applications before you run them.

Link-Editing ILC Applications

When mixing languages within an application, present the desired main routine to the linkage editor first to nominate it as a main routine. You can specify only one main routine.

Note: To get LE/VSE's ILC support in using pre-LE/VSE ILC applications, you must:
- Recompile any C programs with the C/VSE compiler
- Recompile any PL/I programs with the PL/I VSE compiler
- Recompile any DOS/VS COBOL programs with the COBOL/VSE compiler
- Link-edit these applications to replace old HLL library routines with new LE/VSE-conforming routines

Note!

The RPG programming language is *not* a Language Environment-conforming language. You can therefore *not* use RPG directly with other Language Environment-conforming languages that support ILC calls.

For more information, see *LE/VSE Writing Interlanguage Communication Applications* and the migration guide for your primary HLL. (Refer to "Where to Find More Information" on page xxi for information on the various migration guides).

Checking Which Run-Time Options Are in Effect

Using the LE/VSE run-time option RPTOPTS, you can control whether a report is produced; with the LE/VSE run-time option MSGFILE, you can control where report output is directed. RPTOPTS generates a report of all the run-time options that are in effect when your application begins to run. The IBM-supplied default

for RPTOPTS is OFF, meaning a report is not generated when your application finishes running. If you override the default setting of RPTOPTS, a report is sent to the default report destination, SYSLST.

If you want to change the options report destination, you can alter the default setting of the MSGFILE run-time option, which specifies where all run-time diagnostics and messages are written. For example, if you specify MSGFILE(OPTRPRT), the storage report is written to a file whose *filename* is OPTRPRT. Note that you need to provide JCL for OPTRPRT at run time.

For the syntax of RPTOPTS and MSGFILE, see *LE/VSE Programming Reference*

COBOL Considerations

The following discussion pertains to LE/VSE resident routines for callable services.

For COBOL CALL literal statements, the compiler allows you to specify whether your program uses *static* or *dynamic* calls to LE/VSE callable services (or other subroutines):

- When a COBOL program makes a static call to an LE/VSE callable service, the LE/VSE resident routine (or callable service stub) is link-edited with the program.
- When a COBOL program makes a dynamic call to an LE/VSE callable service, the LE/VSE resident routine is not link-edited with the program.

With the exception of certain LE/VSE date and time callable services that can be dynamically called from VS COBOL II programs (see *LE/VSE Programming Reference*), only COBOL programs compiled with the COBOL/VSE compiler can call LE/VSE callable services. COBOL CALL identifier statements are always dynamic calls.

For more information about COBOL static and dynamic calls, see *IBM COBOL for VSE/ESA Programming Guide*

COBOL Linking Considerations

If you relink-edit a VS COBOL II NORES program with LE/VSE, you should code a linkage editor INCLUDE statement for IGZENRI prior to the link-edit step. This will ensure the correct versions of the library routines are included. If you do not include IGZENRI, the link-edited phase will be unnecessarily large and the linkage editor will produce messages such as the following indicating duplicate sections:

```
2139I DUPLICATE SECTION DEFINITION: IGZCDSP .  ***** SECTION IGNORED
```

PL/I Linking Considerations

This section discusses what you need to know if you link in PL/I.

Link-Editing Fetchable Phases

The PL/I FETCH statement dynamically loads a separate phase that can be subsequently invoked from the PL/I procedure that fetches the phase, and the PL/I RELEASE statement deletes the phase. There are some restrictions on the PL/I statements that can be used in fetched procedures. These are described in *IBM PL/I for VSE/ESA Language Reference*

Fetchable (or dynamically loaded) phases should be link-edited into a sublibrary that is subsequently made available as a search sublibrary by means of a LIBDEF PHASE,SEARCH statement.

The job that link-edits a fetchable phase into a sublibrary requires the following linkage editor control statements:

- An ENTRY statement to define the entry point into the PL/I procedure.
- A PHASE statement to define the name used for the fetchable phase. This statement is required if the NAME compile-time option is not used.

  The name by which the fetchable phase is identified in the phase sublibrary must appear in a FETCH or RELEASE statement within the scope of the invoking procedure.

Figure 3 illustrates these statements by showing a job that includes both the compilation and the link-editing of the fetchable PL/I phase.

```
// JOB     jobname
// DLBL    IJSYSLN,'%LEVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,       X
               RECORDS=(400,600)
// DLBL    IJSYS01,'%LEVSE.WORKFILE.IJSYS01',0,VSAM,RECSIZE=4096,      X
               RECORDS=(50,100),DISP=(NEW,KEEP)
// LIBDEF  OBJ,SEARCH=(userlib.sublib,lelib.sublib)
// LIBDEF  PHASE,CATALOG=userlib.sublib
// OPTION  CATAL
   PHASE   FETCH1,*
// EXEC    IEL1AA
    .
    .
    .
    PL/I source(fetchable)
    .
    .
    .
/*
   ENTRY   procedure-name
// EXEC    LNKEDT
/*
/&
```

Figure 3. Example of Link-Editing a Fetchable Phase

LE/VSE-conforming COBOL or C phases can be loaded dynamically by the PL/I FETCH statement.

Fetching Phases with Different AMODEs

LE/VSE supports the PL/I FETCH/RELEASE facility. No special considerations apply to this support when both the fetching phase and the fetched phases have the AMODE(ANY) attribute or both have the AMODE(24) attribute.

LE/VSE also supports the fetching of a phase that has a different AMODE attribute than the phase issuing the FETCH instruction. LE/VSE performs the AMODE switches in this case, and the following constraints apply:

- If any fetched phase is to execute in 24-bit addressing mode, the fetching phase must be loaded into storage below 16MB, and therefore must have the RMODE(24) attribute regardless of its AMODE attribute.
- Any variables passed as parameters to a fetched procedure must be addressable in the AMODE of the fetched procedure. For any fetched phase that is to be executed in 24-bit addressing mode, you must ensure that:

- If any parameter resides in a HEAP area, the BELOW suboption of the HEAP option is specified.
- If any parameter resides in STATIC storage of the fetching phase, the fetching phase has the RMODE(24) attribute so that its STATIC storage is below 16MB.
- If any parameter resides in AUTOMATIC storage, no special considerations apply. If the first two constraints cause problems, then you can copy the variable to a like variable with the AUTOMATIC attribute and pass the copy to the fetched AMODE(24) procedure, with the BELOW suboption of the HEAP option specified.

When a PL/I procedure fetches another PL/I procedure, it is possible for a condition to arise in the fetched procedure for which a PL/I ON-unit was established in the fetching procedure.

PL/I imposes the restriction that if an ON-unit is established while the current addressing mode is 24-bit, and the condition is raised while the addressing mode is 31-bit, the ON-unit is not invoked. This is because PL/I must invoke the ON-unit in the addressing mode in which it was established. If the ON-unit was established in 24-bit addressing mode but the condition arose in 31-bit addressing mode, the code and data required to process the error might not even be addressable in 24-bit addressing mode.

C AMODE/RMODE Considerations

The C run-time library must be addressed in 31-bit addressing mode. Therefore, phases containing C modules must have AMODE=31.

The following table shows valid AMODE and RMODE combinations when your application contains C modules.

Product	RMODE	AMODE	Notes
C only	24 or ANY	31	All programs must use the same AMODE and RMODE combination.
C with COBOL	24 or ANY	31	For VS COBOL II, all COBOL programs must be compiled with the RES and RENT compiler options, which causes AMODE=31. For COBOL/VSE, all COBOL programs must be compiled with the RMODE(ANY) compiler option.
C with PL/I	24 or ANY	31	All programs must use the same AMODE and RMODE combination.
C with CICS	ANY	31	All programs must use this AMODE and RMODE combination.
C with DL/I	24 or ANY	31	All programs must use the same AMODE and RMODE combination.
C with DB2	24 or ANY	31	All programs must use this AMODE and RMODE combination.

For information on AMODE switching, see *LE/VSE C Run-Time Library Reference* or *LE/VSE C Run-Time Programming Guide*

Chapter 3. Prelinking an Application

This chapter describes how to prelink your programs under LE/VSE. The LE/VSE prelinker performs mapping of names, manages writable static areas, collects initialization information, and combines the object modules that form an application into a single object module that can be link-edited for execution.

Which Programs Need to Be Prelinked

You need to prelink C programs compiled with either
- RENT
- LONGNAME

When prelinking, you do not need to include object modules that do not refer to writable static (produced by the RENT compiler option) and do not contain L-names (longnames produced by the LONGNAME compiler option). You could, however, get prelinker warning messages about unresolved references. Any unresolved references will be resolved in the following link-edit step.

When prelinking and link-editing a phase that contains both object modules that need to be prelinked and object modules that do not need to be prelinked, all object modules that require prelinking must be prelinked into a single object module before being link-edited with the object modules that do not require prelinking.

What the Prelinker Does

The prelinker performs the following functions:
- Collects information for run-time initialization, including data initialization for C
- For C object modules compiled with RENT, the prelinker:
  - Combines writable static initialization information
  - Assigns relative offsets to objects in writable static storage
  - Removes writable static name and relocation information
- For programs containing L-names, the prelinker maps L-names to S-names on output (L-names are mixed-case external names, of up to 255 characters in length, emitted by the compiler when compiling with the LONGNAME option; S-names are eight character, single-case external names, emitted by the compiler when compiling with the NOLONGNAME option)

Prelinking Process

Input to the prelinker includes the following:
- Primary input:
  - Control statements and object modules read from SYSIPT
  - Object modules and linkage editor control statements read from SYSLNK
  - Input specified in one or more INCLUDE or LIBRARY control statements processed as primary input

  The prelinker first reads all input from SYSIPT and then reads input from SYSLNK. Input from SYSIPT must be terminated with a /* statement. When reading input from SYSLNK, the prelinker interprets any linkage editor INCLUDE control statements it reads as prelinker INCLUDE control statements,

© Copyright IBM Corp. 1991, 2005

and attempts to resolve them as describe below. All other linkage editor control statements read from SYSLNK are passed to the linkage editor unchanged.

- Secondary input:
  - Object modules read from VSE librarian sublibraries by automatic library call
  - Input specified in one or more INCLUDE or LIBRARY control statements processed as secondary input

The prelinker attempts to resolve each INCLUDE or LIBRARY control statement by reading the named object module from the specified VSE librarian sublibrary or the specified sequential file. The request is resolved if the read is successful.

After the prelinker processes all its input, it writes the prelinked output object module to SYSLNK or SYSPCH, depending on the prelinker options specified. The resultant prelinked object module can then be link-edited.

Using the Prelinker Automatic Library Call

If the prelinker AUTO option is in effect, the prelinker automatic library call is used to resolve referenced and currently undefined symbols (also known as unresolved external references). An undefined symbol becomes a defined symbol if it is contained in the input from the automatic library call process, or from some subsequent input.

If the undefined symbol is an S-name, for example `SNAME`, the VSE librarian sublibraries in the object search chain (specified using the LIBDEF JCL statement) are searched for the member with the same name as the undefined S-name. If the symbol is not the name of an existing VSE librarian member, the symbol can subsequently be defined if a function or variable with the same name is encountered. Unresolved requests generate error or warning messages in the prelinker map.

If the symbol is an L-name that was not resolved by automatic library call and for which a RENAME statement with the SEARCH option is specified, the symbol is resolved under the S-name on the RENAME statement by automatic library call. See “RENAME Control Statement” on page 16 for a complete description of the RENAME control statement.

Writable static references that are not resolved by the prelinker cannot be resolved later. Only the prelinker can be used to resolve writable static. The output object module of the prelinker should not be used as input to another prelink.

LE/VSE Prelinker Map

The LE/VSE prelinker produces a listing file called the prelinker map when you use the MAP prelinker option (which is the default). The prelinker map contains several individual sections that are only generated if they are applicable.

```
========================================================================
|                           PRELINKER MAP               1              |
|                                                                      |
| CPLINK: 5686094 V1 R4 M00 IBM LE/VSE           1996/12/13 12:15:12 |
========================================================================

COMMAND OPTIONS. . . . . : AUTO     NOMEMORY ER       DUP      MAP
                         : NODECK   OBJECT   NONAME   NOUPCASE

========================================================================
|                   OBJECT RESOLUTION WARNINGS       2                  |
========================================================================

WARNING EDC4015: UNRESOLVED REFERENCES ARE DETECTED:
get_caller

========================================================================
|                             FILE MAP                 3              |
========================================================================

*ORIGIN  FILE ID  FILE NAME

   P      00001   DD:SYSLNK
   PI     00002   DD:SYSLIB(GETCALL.OBJ)
   A      00003   DD:SYSLIB(CEEBETBL.OBJ)
   A      00004   DD:SYSLIB(CEEROOTA.OBJ)
   A      00005   DD:SYSLIB(CEESG003.OBJ)
   A      00006   DD:SYSLIB(EDCINPL.OBJ)
   A      00007   DD:SYSLIB(CEEBINT.OBJ)
   A      00008   DD:SYSLIB(CEEBLLST.OBJ)
   A      00009   DD:SYSLIB(CEEBTRM.OBJ)
   A      00010   DD:SYSLIB(CEEBPUBT.OBJ)
   SI     00011   DD:SYSLIB(CEEBPIRA.OBJ)
   A      00012   DD:SYSLIB(CEEARLU.OBJ)
   R      00013   DD:SYSLIB(SUBPRLK.OBJ)
   L      00014   DD:SYSLIB(PRINTF.OBJ)

*ORIGIN:  P=PRIMARY INPUT     PI=PRIMARY INCLUDE    SI=SECONDARY INCLUDE
          A=AUTOMATIC CALL     R=RENAME CARD         L=C LIBRARY
         IN=INTERNAL

========================================================================
|                        WRITABLE STATIC MAP           4              |
========================================================================

  OFFSET    LENGTH  FILE ID  INPUT NAME

       0         4   00001   this_int_is_in_writable_static
       8         C   00001   @STATIC
```

Figure 4. Prelinker Map (Part 1 of 2)

```
========================================================================
|               ESD MAP OF DEFINED AND LONG NAMES      5               |
========================================================================

                 OUTPUT
*REASON  FILE ID  ESD NAME   INPUT NAME

   P      00001   CEESTART   CEESTART
   P      00001   CEEMAIN    CEEMAIN
          00003   CEEBETBL   CEEBETBL
          00004   CEEROOTA   CEEROOTA
   D      00001   MAIN       main
   D      00001   GET@DATE   get_date
   D      00001   GET@NAME   get_name
   P      00005   CEESG003   CEESG003
   R      00013   SUBPRLK    get_year
   L      00014   PRINTF     printf
   D              GET@CALL   get_caller
   D      00001   THIS@INT   this_int_is_not_in_writable_static
   D      00001   @ST00002   Name_Collision_In_First_8
   D      00001   @ST00001   Name_Collision_In_First_Eight
   P      00006   EDCINPL    EDCINPL
   D      00002   GETCALLE   getcaller
          00007   CEEBINT    CEEBINT
          00008   CEEBLLST   CEEBLLST
          00009   CEEBTRM    CEEBTRM
          00010   CEEBPUBT   CEEBPUBT
          00011   CEEINT     CEEINT
          00012   CEEARLU    CEEARLU
          00008   CEELLIST   CEELLIST
          00011   CEEBPIRA   CEEBPIRA
          00011   CEEBPIRB   CEEBPIRB
          00011   CEEBPIRC   CEEBPIRC
          00011   CEECPYRT   CEECPYRT
          00014   PRINTF     PRINTF

*REASON: P=#PRAGMA OR RESERVED    S=MATCHES SHORT NAME    R=RENAME CARD
         L=C LIBRARY              U=UPCASE OPTION         D=DEFAULT


=========== E N D   O F   P R E - L I N K A G E   M A P ============
```

Figure 4. Prelinker Map (Part 2 of 2)

The numbers in the following text correspond to the numbers shown in the map.

1 Heading
The heading is always generated and contains the product number, the library release number, the library version number, the date and the time the prelink step began, followed by a list of the prelinker options in effect for the step.

2 Object Resolution Warnings
This section is generated if objects remained undefined at the end of the prelink step or if duplicate objects were detected during the step. The names of the applicable objects are listed.

3 File Map
This section lists the object modules that were included in input. An object module consisting only of RENAME control statements, for example, is *not* shown. Also provided in this section are object module origin (`*ORIGIN`), identifier (`FILE ID`), and name (`FILE NAME`) information. `*ORIGIN` indicates the reason the object module was included:

- The object module was read from SYSIPT or SYSLNK

- An INCLUDE control statement in primary or secondary input
- A RENAME control statement
- Resolution of library references by automatic library call
- The object module was internal and self-generated by the prelink step

The `FILE ID` can be found in other sections and is used as a cross-reference to the object module.

The `FILE NAME` can be either the member name, the VSE librarian sublibrary and member names, or the sequential file filename.

4 Writable Static Map
This section is generated if an object module was encountered that contains defined static external data. This section lists the name of each symbol, the length, the relative offset within the writable static area, and a `FILE ID` for the file containing the symbol's definition.

5 ESD Map of Defined and Long names
This section lists the names of external symbols that are not in writable static. It also shows a mapping of input L-names to output S-names.

If the object is defined, the `FILE ID` indicates the file that contains the definition. Otherwise, this field is left blank. For any name, the input name and output S-name are listed. If the input name is an L-name, the rule used to map the L-name to the S-name is applied. If the name is not an L-name, this field is left blank.

Control Statement Processing

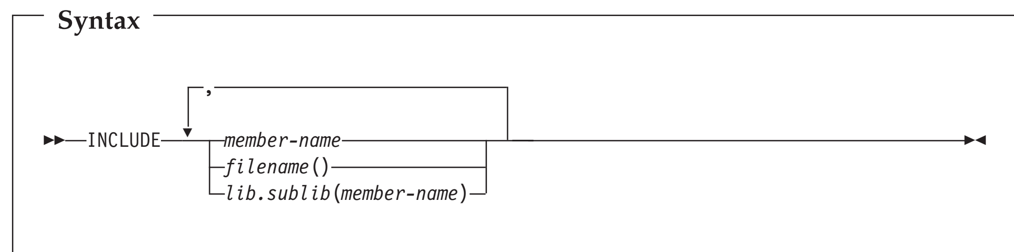
The only job control statements processed by the prelinker are INCLUDE, LIBRARY, and RENAME. The remaining control statements are left unchanged until the link step.

Note: If you cannot fit all of the information on one control statement, you can use one or more continuations. The L-name, for example, can be split across more than one statement. Continuations are enabled by placing a nonblank character in column 72 of the statement that is to be continued. They must begin in column 16 of the next statement.

INCLUDE Control Statement

The INCLUDE control statement indicates that an object module or further control statements are to be included for processing by the prelinker. The INCLUDE control statement has the following syntax:

Syntax



member-name
The name of the object module to be included. The object module must be a member in a VSE librarian sublibrary. If the sublibrary is not specifically

identified by *lib.sublib*, the sublibrary must be included in the object search chain as defined by the LIBDEF JCL statement.

filename
The filename of a sequential disk or tape file containing the object module to be included. The filename must match a filename used in a DLBL or TLBL JCL statement. The sequential file must be fixed-length unblocked 80 character records.

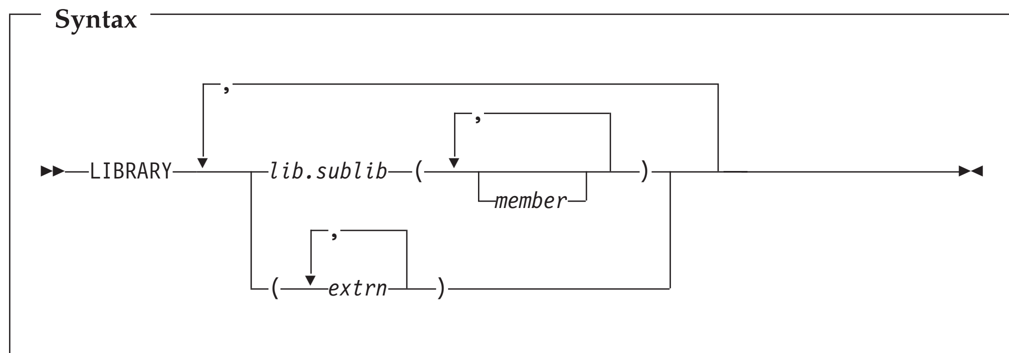
lib.sublib
The name of a VSE librarian sublibrary containing the object module to be included.

Duplicate INCLUDE statements are ignored.

LIBRARY Control Statement

The LIBRARY control statement is used to specify the names of additional VSE librarian sublibraries to be searched by automatic library call processing, or the names of external references that are not to be resolved by automatic library call processing. The LIBRARY control statement has the following syntax:

Syntax



lib.sublib
The name of a VSE librarian sublibrary.

member
The member name of an object module in the specified sublibrary. Only S-names can be specified, and the name must match the name of the unresolved external reference. If an L-name is specified, the external reference will not be resolved from the specified sublibrary. No warning will be issued.

Automatic library calls search for the specified member in the specified sublibrary instead of the sublibraries in the object search chain.

extrn
An external reference that could be unresolved after primary input processing. This external reference will not be resolved by an automatic library call. Only S-names can be specified. If an L-name is specified, the prelinker will still attempt to resolve the external reference. No warning will be issued.

The LIBRARY control statement is removed and not placed in the prelinker output object module; the linkage editor does not see the LIBRARY control statement.

RENAME Control Statement

The RENAME control statement has the following syntax:

Syntax

```
►►──RENAME──L-name──S-name──┬────────┬──────────────────────────►◄
                            └─SEARCH─┘
```

L-name
: The name of the input L-name to be renamed on output. All occurrences of this L-name are renamed.

S-name
: The name of the output S-name to which the L-name will be changed. This name can be at most 8 characters and case is respected.

SEARCH
: An optional parameter specifying that if the S-name is undefined, the prelinker searches by an automatic library call for the definition of the S-name.

The RENAME control statement is processed by the prelinker and can be used for several purposes:

- To explicitly override the default name given to an L-name when an L-name is mapped to an S-name.

  You can explicitly control the names presented to the system linkage editor so that external variable and function names are consistent from one linkage editor run to the next. This consistency makes it easier to recognize control section and label names that appear in system dumps and linkage editor listings. Another mapping rule (described in “Mapping L-Names to S-Names” on page 18) can provide the suitable name, but if you need to replace the linkage editor control section, you need to maintain consistent names.
- To explicitly bind an L-name to an S-name. This binding might be necessary when communicating with objects from other language and assembler processors, because these processors generate only S-names.
- A RENAME control statement cannot be used to rename a writable static object because its name is not contained in the output from the prelinker.

RENAME control statements can be placed before, between, or after other control statements or object modules. An object module can contain only RENAME statements. Also, RENAME statements can be placed in input that is included because of other RENAME statements.

Usage Notes

- A RENAME statement is ignored if the L-name is not encountered in the input.
- A RENAME statement for an L-name is valid provided **all** of the following are true:
  - The L-name was not already mapped because of a rule that preceded the RENAME statement rule in the hierarchy described in “Mapping L-Names to S-Names” on page 18.
  - The L-name was not already mapped because of a previous valid RENAME statement for the L-name.
  - The S-name is not itself an L-name. This rule holds true even if the S-name has its own RENAME statement.

– A previous valid RENAME statement did not rename another L-name to the same S-name.
– Either the L-name or the S-name is not defined. Either the L-name or the S-name can be defined, but not both. This rule holds true even if the S-name has its own RENAME statement.

Mapping L-Names to S-Names

The output object module of the prelinker can be used as input to the system linkage editor.

Because the system linkage editor accepts only S-names, the LE/VSE prelinker maps L-names to S-names on output. S-names are not changed. L-names can be up to 255 characters in length; truncation of the L-names to the 8-character S-name limit is therefore not sufficient because collisions can occur.

The LE/VSE prelinker maps a given L-name to an S-name according to the following hierarchy:

1. If any occurrence of the L-name is a reserved run-time name, or was caused by a `#pragma map` or `#pragma CSECT` directive, then that same name is chosen for all occurrences of the name. This name must not be changed, even if a RENAME control statement for the name exists. For information on the RENAME control statement, see “RENAME Control Statement” on page 16.
2. If the L-name was found to have a corresponding S-name, the same name is chosen. For example, `DOTOTALS` is coded in both a C and assembler program. This name must not be changed, even if a RENAME statement for the name exists. This rule binds the L-name to its S-name.
3. If a valid RENAME statement for the L-name is present, the S-name specified on the RENAME statement is chosen.
4. If the name corresponds to an LE/VSE function or library object for which you did not supply a replacement, the name chosen is the truncated, uppercased version of the L-name library name (with _ mapped to @).

   The S-name is not chosen, if either:

   - A valid RENAME statement renames another L-name to this S-name. For example, the RENAME statement `RENAME mybigname PRINTF` would make the library `printf()` function unavailable if `mybigname` is found in input.
   - Another L-name is found to have the same name as the S-name. For example, explicitly coding and referencing `SPRINTF` in the C source program would make the library `sprintf()` function unavailable.

   Avoid such practices to ensure that the appropriate LE/VSE function is chosen.
5. If the UPCASE option is specified, names that are 8 characters or fewer are changed to uppercase (with _ mapped to @). Names that begin with `IBM` or `CEE` will be changed to `IB$`, and `CE$`, respectively. Because of this rule, two different names can map to the same name. You should therefore use the UPCASE option carefully. A warning message is issued if a collision is found, but the names are still mapped.
6. If none of the above rules apply, a default mapping is performed. This mapping is the same as the one the compiler option NOLONGNAME uses for external names, taking collisions into account. That is, the name is truncated to 8 characters and changed to uppercase (with _ mapped to @). Names that begin with `IBM` or `CEE` will be changed to `IB$` and `CE$`, respectively. If this name is the same as the original name, it is always chosen. This name is also chosen if a name collision does not occur. A name collision occurs if either

- The S-name has already been seen in **any** input, that is, the name is not new.
- After applying this default mapping, the same name is generated for at least two, previously unmapped, names.

If a collision occurs, a unique name is generated for the output name. For example, the name `@ST00033` is manufactured.

A program that is compiled with the NOLONGNAME compiler option and link-edited, except for collisions, presents the linkage editor with the same names as when the program is compiled with the LONGNAME option and processed by the prelinker.

Running the Prelinker

The following is sample JCL to run the prelinker.

```
// JOB PRELINK
// LIBDEF PHASE,CATALOG=(USER.SUBLIB)
// LIBDEF OBJ,SEARCH=(USER.SUBLIB,PRD2.SCEEBASE)
// DLBL IJSYSLN,'%LEVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,      X
               RECORDS=(400,600)
// OPTION  CATAL
   ACTION  MAP
   PHASE   PROGRAM1,*
⋮
compilation step
⋮
 INCLUDE GETCALL
// EXEC EDCPRLK,SIZE=EDCPRLK,PARM='prelinker options'
 RENAME get_year SUBPRLK SEARCH
/*
// EXEC LNKEDT
/&
```

Figure 5. JCL for Prelinking a C Program

The prelinker is actually an LE/VSE-conforming C program, and requires LE/VSE to run. If you want to specify LE/VSE run-time options when invoking the prelinker, you should code an EXEC statement in the format shown below:

```
// EXEC EDCPRLK,SIZE=EDCPRLK,PARM='run-time options/prelinker options'
```

For more information about specifying LE/VSE run-time options in the EXEC statement, see “Specifying Run-Time Options in the EXEC Statement” on page 30.

Prelinker Options

The following table describes the LE/VSE prelinker options.

Table 9. Prelinker Options

Option	Description
AUTO | NOAUTO	AUTO specifies that the prelinker should try to resolve unresolved short name references by searching all sublibraries in the object search chain for OBJ files of the same name.
DECK | NODECK	DECK specifies that the prelinker is to write the prelinked object module to SYSPCH. The DECK option is specified using the JCL OPTION statement. This option cannot be specified in the PARM parameter of the JCL EXEC statement.

Table 9. Prelinker Options (continued)

Option	Description
DUP | NODUP	DUP specifies that if duplicate symbols are detected, the symbol names should be directed to `stdout`, and the return code minimally set to a warning level of 4. NODUP does not affect the return code setting when duplicates are detected.
ER | NOER	ER specifies that if there are unresolved references, a message and list of unresolved symbols are written as part of the prelinker map. For unresolved references, the return code is minimally set to warning level 4. For unresolved writable static references, the return code is minimally set to error level 8. NOER specifies that a list of unresolved symbols is not written as part of the prelinker map. For unresolved references, the return code is unaffected. For unresolved writable static references, the return code is minimally set to warning level 4.
MAP | NOMAP	The MAP option specifies that the prelinker should generate a prelink listing. See "LE/VSE Prelinker Map" on page 13 for a description of the map.
MEMORY | **NOMEMORY**	The MEMORY option specifies that the prelinker will buffer (retain in storage), for the duration of the prelink step, those object modules that are read and processed. The MEMORY option is used to increase prelinker speed. To use this option, however, you might require additional memory. If you use this option and the prelink fails due to a storage error, you must increase your storage size or use the prelinker without the MEMORY option.

NAME(*name*) | **NONAME**	NAME is used to generate a linkage editor PHASE statement or a VSE librarian CATALOG statement.
	
	When the NAME option is specified in conjunction with the OBJECT option, the prelinker writes a linkage editor PHASE statement to SYSLNK for input to the linkage editor. The format of the PHASE statement is: is as follows:
	`PHASE name,*`
	
	When the NAME option is specified in conjunction with the DECK option, the prelinker writes a VSE librarian CATALOG statement to SYSPCH. This can be used as input to the librarian to catalog the object code in a VSE librarian sublibrary. The format of the CATALOG statement is:
	`CATALOG name.OBJ REPLACE=YES`

OBJECT | NOOBJECT	OBJECT specifies that the prelinker is to write the prelinked object module to SYSLNK for input to the linkage editor. The OBJECT option is specified using the LINK or CATAL options of the JCL OPTION statement. This option cannot be specified in the PARM parameter of the JCL EXEC statement.
UPCASE | **NOUPCASE**	The UPCASE option enforces the uppercase mapping of those L-names that are 8 characters or fewer and have not been explicitly mapped by another mechanism. These L-names will be uppercased (with _ mapped to @), and names that begin with `IBM` or `CEE` will be changed to `IB$` and `CE$`, respectively. The UPCASE option is useful when calling routines written in languages other than C. For example, COBOL and assembler each uppercase all of their external names. So, if the names are coded in lowercase in the C program and the LONGNAME option is used, the names will not match by default. The UPCASE option can be used to enforce this matching. The RENAME control statement can also be used for this purpose.

Chapter 4. Linking and Running

You process an application by submitting batch jobs to the VSE operating system. A job might consist of one or more of the following job steps:
- Compiling a program
- Link-editing an application
- Running an application

The following section provides an overview of link-editing and running LE/VSE-conforming applications. For detailed information about link-editing, refer to *z/VSE System Control Statements* For information about the LE/VSE prelinker, see Chapter 3, "Prelinking an Application," on page 11.

Basic Linking and Running

This section describes how to accept the default LE/VSE run-time options, and how to override the default LE/VSE run-time options using your JCL.

Linking and Running an Existing Object Module

To run an existing object module and accept all of the default LE/VSE run-time options, use the following sample JCL.

```
// JOB     jobname
// DLBL    IJSYSLN,'%LEVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,       X
               RECORDS=(400,600)
// LIBDEF  OBJ,SEARCH=(userlib.sublib,lelib.sublib)
// LIBDEF  PHASE,SEARCH=(lelib.sublib)
// OPTION  LINK
   INCLUDE objmod
// EXEC    LNKEDT
// EXEC
/*
/&
```

Figure 6. Accepting the Default Run-Time Options

Overriding the Default Run-Time Options

In the following example, an object module called MYPROG is link-edited and run. The JCL in the example overrides the LE/VSE defaults for the RPTOPTS and MSGFILE run-time options.

© Copyright IBM Corp. 1991, 2005

```
// JOB     jobname
// DLBL    IJSYSLN,'%LEVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,       X
               RECORDS=(400,600)
// DLBL    OPTRPRT,'fileid',0,SD
// EXTENT  SYSnnn,volser,1,0,start,tracks
// ASSGN   SYSnnn,DISK,VOL=volser,SHR
// LIBDEF  OBJ,SEARCH=(userlib.sublib,lelib.sublib)
// LIBDEF  PHASE,SEARCH=(lelib.sublib)
// OPTION  LINK
   INCLUDE MYPROG
// EXEC    LNKEDT
// EXEC    ,PARM='RPTOPTS(ON),MSGFILE(OPTRPRT)/'
/*
/&
```

Figure 7. Overriding the Default Run-Time Options

The trailing slash after the run-time options is required when MYPROG is a C or PL/I program. It is used to terminate the run-time options and separate them from any program arguments that follow. However, when MYPROG is a COBOL program, LE/VSE, by default, expects program arguments to be specified before run-time options. In this case, you need to code the slash before the run-time options. Alternatively, you can use the CBLOPTS run-time option to change the order in which LE/VSE expects program arguments and run-time options to be specified. Therefore, when MYPROG is a COBOL/VSE program, you need either to specify the CBLOPTS(OFF) run-time option at installation (see *LE/VSE Customization Guide*), or code the following:

```
⋮
// EXEC    ,PARM='/RPTOPTS(ON),MSGFILE(OPTRPRT)'
⋮
```

Figure 8. Overriding the Default Run-Time Options for COBOL

Note: When running under CICS, DL/I, or DB2, you cannot override default LE/VSE run-time options using your JCL. You need to assemble an application defaults module, CEEUOPT, and link-edit it with your application. For more information, see Chapter 5, “Using Run-Time Options,” on page 33.

Providing Input to the Linkage Editor

The linkage editor processes your compiled program (object module) and readies it for execution. The processed module becomes an executable phase.

Input to the linkage editor can be:
- One or more object modules
- Linkage editor control statements

Figure 9 on page 23 shows the basic link-editing process for your application.

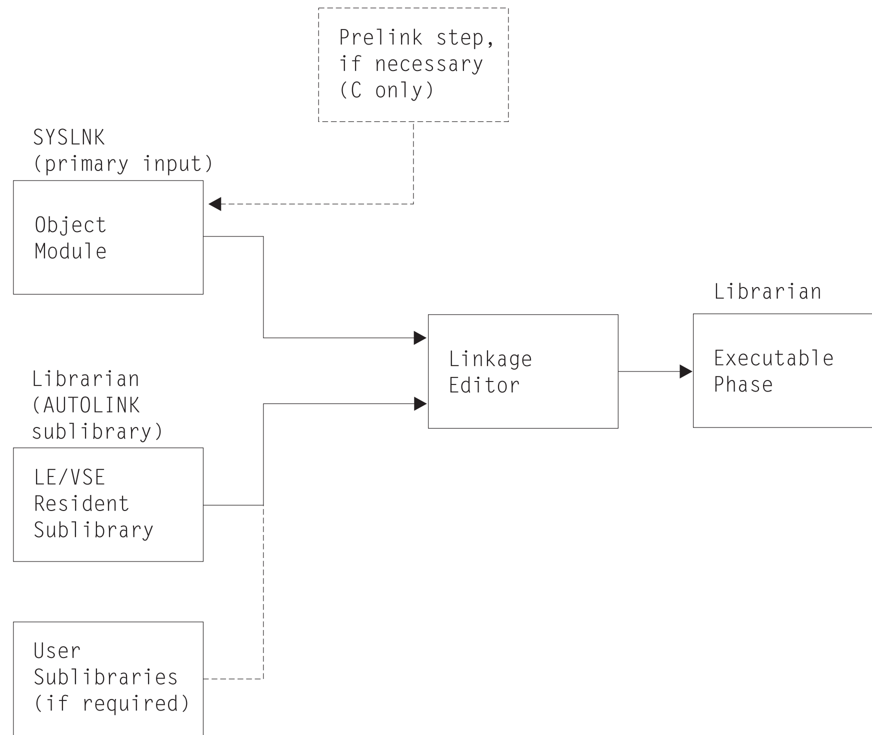


Figure 9. Basic Linkage Editor Processing

Writing JCL for the Linkage Editor

JCL for the linkage editor is required to:
- Specify the files used by the linkage editor
- Specify the linkage editor control statements
- Invoke the linkage editor

The following sections provide a brief explanation of the JCL required.

Specifying the Files Used by the Linkage Editor

Table 10 shows the files used by the linkage editor, and summarizes the function and permissible device types for each file.

Table 10. Files Used for Link-Editing

File	Type	Function	Permissible Device Types
SYSIPT[1]	Input	Additional object module input	Card reader Magnetic tape Direct access
SYSLNK	Input	Object module input, normally the output of the compiler or the prelinker	Direct access
SYSLST[2]	Output	Diagnostic messages Informative messages Linkage editor map	Printer Magnetic tape Direct access
SYSLOG	Output	Operator messages	Display console
SYSRDR	Input	Control statement input	Card reader Magnetic tape Direct access
IJSYS01 (SYS001)	Work file	Linkage editor work file	Direct access

Table 10. Files Used for Link-Editing (continued)

File	Type	Function	Permissible Device Types
User-specified Sublibrary	Library	Catalog sublibrary for the phase[3] External reference and INCLUDE statement resolution[4]	Direct access

Notes:
1. Object modules read from SYSIPT are written to SYSLNK
2. If not provided, messages are written to SYSLOG
3. Required if the phase is to be cataloged
4. Required for additional object module input

Your installation will probably have standard labels in place for most of the files used for link-editing, in which case you do not need to provide JCL for them. Check with your system administrator.

Specifying Linkage Editor Control Statements

In addition to object modules, input to the linkage editor includes linkage editor control statements. These statements are briefly described in Table 11.

Table 11. Linkage Editor Control Statements

Statement	Action	Comments
ACTION	Use the ACTION statement to specify linkage editor options. The options that may be specified are: • MAP—requests the linkage editor to write a linkage editor map to SYSLST. • NOMAP—suppresses the MAP option. • NOAUTO—suppresses the automatic library lookup (AUTOLINK) function; the linkage editor will not attempt to resolve external references using the automatic library lookup feature. • CANCEL—requests the linkage editor to cancel the job if a linkage editor error occurs. • SMAP—request the linkage editor to produce a sorted listing of CSECT names on SYSLST.	This statement, if present, must be the first linkage editor statement in your input stream. ACTION MAP is the default, if SYSLST is assigned.
ENTRY	Use the ENTRY statement to specify the entry point of a phase that has multiple possible entry points.	The default entry point is the first significant address the linkage editor encounters in an END record in an object module or, if none is found, the load address of the phase.
INCLUDE	Use the INCLUDE statement to include additional object modules in the phase that would not otherwise be included.	You can use the INCLUDE statement to include an object module that was cataloged with a different name than the name used in the CALL statement in your program.

Table 11. Linkage Editor Control Statements (continued)

Statement	Action	Comments
MODE	Use the MODE statement to specify the addressing mode and the residence mode of a phase. The addressing mode and residence mode can be specified by the following options of the MODE statement: • AMODE(24 | 31 | ANY)—requests the linkage editor to override the default AMODE attribute established by the compiler. • RMODE(24 | ANY)—requests the linkage editor to override the default RMODE attribute established by the compiler.	This statement must follow the PHASE statement. Alternatively, the AMODE and RMODE parameters may be specified in the PARM parameter of the EXEC LNKEDT statement.
PHASE	Use the PHASE statement to provide the linkage editor with a phase name.	You must provide a PHASE statement (and the job control option CATAL) if you wish to catalog the phase in a VSE Librarian sublibrary.

Using the EXEC Statement

Use the EXEC job control statement in your JCL to invoke the linkage editor. The EXEC statement to invoke the linkage editor is:

```
// EXEC LNKEDT
```

Using the PARM Parameter for the Linkage Editor

You can use the PARM parameter of the EXEC statement to specify the AMODE and RMODE of your executable phase. Normally, the linkage editor assigns the AMODE and RMODE from the AMODE and RMODE attributes of the input modules, but you can override them. For example, if you want the phase produced by the linkage editor to have RMODE(24), specify:

```
// EXEC LNKEDT,PARM='RMODE=24'
```

Example of Linkage Editor JCL

A typical sequence of job control statements for link-editing an object module into a phase is shown in Figure 10. The PHASE linkage editor control statement in the figure specifies that the link-edited phase is to have the name PROGRAM1. The LIBDEF PHASE and OPTION CATAL job control statements specify that the link-edited phase is to be cataloged in the sublibrary USER.RUNLIB.

```
// JOB     LNKEDT
// DLBL    IJSYSLN,'%LEVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,       X
               RECORDS=(400,600)
// LIBDEF  OBJ,SEARCH=(USER.OBJLIB,PRD2.SCEEBASE)
// LIBDEF  PHASE,CATALOG=(USER.RUNLIB)
// OPTION  CATAL
   ACTION  MAP
   PHASE   PROGRAM1,*
   INCLUDE PROGRAM1
// EXEC    LNKEDT
/*
/&
```

Figure 10. Creating a Phase

Link-Editing Multiple Object Modules

When you link-edit an object module containing any of the following:

A C `main()` function
A C fetchable function
A PL/I MAIN procedure

the entry point of the resultant phase is resolved to the external symbol CEESTART. This happens automatically because the C and PL/I compilers generate a CEESTART CSECT at the beginning of such object modules. Run-time errors occur if the phase entry point is forced to some other symbol by use of the linkage editor ENTRY control statement.

If you link-edit such an object module with object modules produced by the same language compiler, other language compilers, or by assembler, the module containing the CEESTART CSECT must be the first to receive control, and you must ensure that the entry point of the resulting phase is resolved to the external symbol CEESTART. This happens automatically if the object module containing the CEESTART CSECT is first in the input to the linkage editor.

Alternatively, the following linkage editor ENTRY control statement can be included in the input to the linkage editor:

```
ENTRY CEESTART
```

Using the INCLUDE Statement

You can use the INCLUDE linkage editor control statement to specify additional object modules that you want included in the output phase. Figure 11 contains an example of how to link-edit the CEEUOPT CSECT with your application. In the example, CEEUOPT is used to establish application run-time option defaults; see Chapter 5, “Using Run-Time Options,” on page 33 for more information.

```
// DLBL    IJSYSLN,'%LEVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,      X
                 RECORDS=(400,600)
// LIBDEF  OBJ,SEARCH=(USER.OBJLIB,PRD2.SCEEBASE)
// LIBDEF  PHASE,CATALOG=(USER.RUNLIB)
// OPTION  CATAL
   ACTION  MAP
   PHASE   PROGRAM1,*
   INCLUDE CEEUOPT
⋮
compilation step
⋮
// EXEC    LNKEDT
```

Figure 11. Using the INCLUDE Linkage Editor Control Statement

Linkage Editor Module Map

If you specify the ACTION MAP linkage editor control statement, the linkage editor produces:

- A list of the linkage editor control statements and a list all object modules specifically included in the phase, or obtained from a Librarian sublibrary by the automatic library lookup feature. The list includes the name of the object module, the date and time the object module was first cataloged, the date and time of the latest update to the module, and the name of the sublibrary from which the module was included.

- A map of the link-edited phase that shows all control sections in the phase and all the entry point names in each control section. Named common areas are listed as control sections.

  For each control section, the map indicates its origin (as an address within the partition in which the link-edit is run), its offset from the beginning of the partition, and its offset within the phase. For each entry name in each control section, the map indicates its location (as an address within the partition in which the link-edit is run).

  The control sections are arranged in ascending order according to their origin. An entry name (prefixed by + or *) is listed with the control section in which it is defined.

  The entry (or transfer) address, that is, the relative address within the partition of the main entry point, is at the beginning of the map. The entry address is followed by the lowest and highest addresses of the phase within the partition. Pseudoregisters, if used, also appear in the map; the name, length, and displacement of each pseudoregister are given.

Figure 12 contains a linkage editor listing and map showing twelve control sections. There is one named control section (CALLIVP1) and eleven control sections obtained from the LE/VSE library. In addition, four entry names are defined: CEELLIST in CEEBLLST; CEEINT, CEEBPIRB, and CEEBPIRC in CEEBPIRA.

```
ACTION TAKEN  MAP
            ** MODULE CEEBETBL 95-11-23 09.43                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEBINT  95-11-23 09.43                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEBLLST 95-11-23 09.43                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEBPUBT 95-11-23 09.43                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEBTRM  95-11-23 09.43                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEESG005 95-11-22 16.14                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEESTART 95-11-23 09.44                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE IGZCBSN  95-11-22 16.14                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEARLU  95-11-23 09.43                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEINT   95-11-23 09.43                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
INCLUDE CEEBPIRA
            ** MODULE CEEBPIRA 95-11-23 09.43                 INCLUDED FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
ENTRY
12/13/96 PHASE    XFR-AD  LOCORE  HICORE    CSECT/    LOADED   RELOC.  PARTIT. PHASE   TAKEN  AMODE/RMODE
                                            ENTRY     AT       FACTOR  OFFSET  OFFSET  FROM
         ---------------------------------------------------------------------------------------------------
         PHASE*** 500078  500078  5029D1                                                        31 ANY  RELOCATABLE
         ---------------------------------------------------------------------------------------------------
                                            CALLIVP1  500078   500078  000000  000000  SYSLNK   ANY ANY
                                            CEEBETBL  501A90   501A90  001A18  001A18  CEEBETBL ANY ANY
                                            CEEBINT   501AB0   501AB0  001A38  001A38  CEEBINT  ANY ANY
                                            CEEBLLST  501AB8   501AB8  001A40  001A40  CEEBLLST ANY ANY
                                           *CEELLIST  501AC8
                                            CEEBPUBT  501B18   501B18  001AA0  001AA0  CEEBPUBT ANY ANY
                                            CEEBTRM   501B38   501B38  001AC0  001AC0  CEEBTRM  ANY ANY
                                            CEESG005  501E60   501E60  001DE8  001DE8  CEESG005 ANY ANY
                                            CEESTART  501E78   501E78  001E00  001E00  CEESTART ANY ANY
                                            IGZCBSN   501EF8   501EF8  001E80  001E80  IGZCBSN   31 ANY
                                            CEEARLU   502640   502640  0025C8  0025C8  CEEARLU  ANY ANY
                                            CEEBPIRA  5026E0   5026E0  002668  002668  CEEBPIRA ANY ANY
                                           +CEEINT    5026E0
                                           *CEEBPIRB  5026E0
                                           *CEEBPIRC  5026E0
                                            CEECPYRT  5028F0   5026E0  002878  002878  CEEBPIRA ANY ANY
```

Figure 12. Link-Edit Listing and Module Map

Detecting Link-Edit Errors

After link-editing, you receive a listing of diagnostic messages on SYSLST. Check the linkage editor map to ensure that all the object modules you expected were included.

When link-editing your application, you might get unresolved *weak external references* (WXTRN). The example of a link-edit map in Figure 13 shows unresolved weak external references. These unresolved weak external references do not necessarily indicate that there were errors when your application was link-edited. However, all strong external references (EXTRN) should be resolved for your application to run correctly.

```
12/13/96 PHASE    XFR-AD  LOCORE  HICORE    CSECT/    LOADED    RELOC.  PARTIT. PHASE   TAKEN  AMODE/RMODE
                                            ENTRY     AT        FACTOR  OFFSET  OFFSET  FROM
         ------------------------------------------------------------------------------------------------------
         PHASE*** 500078  500078  5029D1                                                        31 ANY  RELOCATABLE
         ------------------------------------------------------------------------------------------------------
                                            CALLIVP1  500078    500078  000000  000000  SYSLNK   ANY ANY
                                            CEEBETBL  501A90    501A90  001A18  001A18  CEEBETBL ANY ANY
                                            CEEBINT   501AB0    501AB0  001A38  001A38  CEEBINT  ANY ANY
                                            CEEBLLST  501AB8    501AB8  001A40  001A40  CEEBLLST ANY ANY
                                           *CEELLIST  501AC8
                                            CEEBPUBT  501B18    501B18  001AA0  001AA0  CEEBPUBT ANY ANY
                                            CEEBTRM   501B38    501B38  001AC0  001AC0  CEEBTRM  ANY ANY
                                            CEESG005  501E60    501E60  001DE8  001DE8  CEESG005 ANY ANY
                                            CEESTART  501E78    501E78  001E00  001E00  CEESTART ANY ANY
                                            IGZCBSN   501EF8    501EF8  001E80  001E80  IGZCBSN   31 ANY
                                            CEEARLU   502640    502640  0025C8  0025C8  CEEARLU  ANY ANY
                                            CEEBPIRA  5026E0    5026E0  002668  002668  CEEBPIRA ANY ANY
                                           +CEEINT    5026E0
                                           *CEEBPIRB  5026E0
                                           *CEEBPIRC  5026E0
                                            CEECPYRT  5028F0    5026E0  002878  002878  CEEBPIRA ANY ANY
UNRESOLVED EXTERNAL REFERENCES                      WXTRN     CEEUOPT
                                                    WXTRN     CEEBXITA
                                                    WXTRN     CEESG000
                                                    WXTRN     CEESG001
                                                    WXTRN     CEESG002
                                                    WXTRN     CEESG003
                                                    WXTRN     CEESG004
                                                    WXTRN     CEESG006
                                                    WXTRN     CEESG007
                                                    WXTRN     CEESG008
                                                    WXTRN     CEESG009
                                                    WXTRN     CEESG010
                                                    WXTRN     CEESG011
                                                    WXTRN     CEESG012
                                                    WXTRN     CEESG013
                                                    WXTRN     CEESG014
                                                    WXTRN     CEESG015
                                                    WXTRN     CEESG016
                                                    WXTRN     CEEMAIN
                                                    WXTRN     CEEFMAIN
                                                    WXTRN     CEEROOTA
                                                    WXTRN     IGZETUN
                                                    WXTRN     IGZEOPT
                                                    WXTRN     ILBDMNS0
UNRESOLVED ADCON  AT OFFSET 00501AA0
UNRESOLVED ADCON  AT OFFSET 00501A94
UNRESOLVED ADCON  AT OFFSET 00501AC8
.
.
.
UNRESOLVED ADCON  AT OFFSET 00502638
024 UNRESOLVED ADDRESS CONSTANTS
```

Figure 13. Link-Edit Map of a COBOL/VSE Program with Unresolved Weak External References

Running an Application

You can request the execution of a phase in an EXEC statement in your JCL. The general form of the EXEC statement is:

```
// EXEC [PGM=]program_name,SIZE=program_size
```

where *program_name* is the name of the phase of the application to be executed and *program_size* is the amount of program storage required to run the application.

Note: The amount of program storage required to run an application does not include the storage required for LE/VSE heap and stack storage, LE/VSE library routines, or dynamically loaded application routines. Program storage is generally only used to load the main program and, if required, the SORT product (and its work areas). LE/VSE uses partition GETVIS storage for all other storage requirements. LE/VSE requires a minimum of 1200KB below-the-line GETVIS storage.

Specifying the Search Order

When you use JCL to request execution of a phase (or when you dynamically load a phase), the specified phase must be one of the following:

- A phase that is resident in the shared virtual area (SVA), and defined in the *system directory list* (SDL)
- A phase that is a member of a private sublibrary specified in a LIBDEF statement in your JCL
- A phase that is a member of a sublibrary specified in the permanent search chain for the partition in which your job is running
- A phase that is a member of the system sublibrary, IJSYSRS.SYSLIB

Unless you specify one or more private libraries in a LIBDEF statement, VSE searches only the SVA, those sublibraries specified in the permanent search chain for the partition in which your job is running, and the system sublibrary for the phase you specify.

You can define a sublibrary in a LIBDEF statement in the following ways:
- With the LIBDEF *,SEARCH statement in your JCL
- With the LIBDEF PHASE,SEARCH statement in your JCL

This sublibrary is searched after the SDL, but before the sublibraries defined in the permanent search chain and before the system sublibrary. If you do not want the SDL to be searched first, your must specify it at the appropriate position in the search chain defined by the LIBDEF statement.

In the following example, assuming the phase PROGRAM1 is not in the SDL, the system searches the private sublibrary USER.RUNLIB for the phase PROGRAM1, reads the phase into storage, and executes it.

```
// JOB     RUNPROG1
// LIBDEF  PHASE,SEARCH=(PRD2.SCEEBASE,USER.RUNLIB)
// EXEC    PROGRAM1,SIZE=PROGRAM1
/*
/&
```

Specifying Run-Time Options

Each time your application runs, a set of run-time options must be established. These options determine many of the properties of how the application runs, including its performance, error handling characteristics, storage management, and production of debugging information. You can specify run-time options in any of the following places:

- In the CEEDOPT CSECT, where the installation default options are located (see “CEEXOPT Invocation Syntax” on page 40 for more information)
- In the CEEUOPT CSECT where user-supplied default options are located (see “CEEXOPT Invocation Syntax” on page 40 for more information)
- `#pragma runopts` in C source code (see page 36 for more information)
- In a PLIXOPT string in PL/I source code (see page 37)
- In the PARM parameter of the EXEC statement in your JCL (see below)
- In the assembler user exit (see “CEEBXITA Assembler User Exit Interface” on page 323 for more information)

Specifying Run-Time Options in the EXEC Statement

You can override those installation default run-time options specified as overridable, and any application default run-time options, by specifying run-time options in the PARM parameter of the EXEC statement. The general form for specifying run-time options in the PARM parameter of the EXEC statement is:

```
// EXEC [PGM=]program_name,                                          X
               PARM='[run-time options/][program parameters]'
```

For example, if you want to generate a storage report and run-time options report for the application PROGRAM1, specify the following:

```
// EXEC PROGRAM1,PARM='RPTSTG(ON),RPTOPTS(ON)/'
```

The run-time options that are passed to the main routine must be followed by a slash (/) to separate them from program parameters. For HLL considerations to keep in mind when specifying run-time options, see “Specifying Run-Time Options and Program Arguments” on page 38. The EXECOPS option for C is used to specify that run-time options passed as parameters at execution time are to be processed by LE/VSE. The option NOEXECOPS specifies that run-time options are not to be processed from execution parameters and are to be treated as program parameters. You can specify either option in a `#pragma runopts` statement in your C program. You can also specify either option as a C/VSE compile-time option. EXECOPS is the default. When EXECOPS is in effect, you can pass run-time options in the EXEC statement in your JCL.

VSE normally limits the size of the string you can specify in the PARM parameter of the EXEC statement to 100 characters. However, if you are running VSE/ESA Version 2 Release 2, or a previous supported release of VSE/ESA with the appropriate PTF applied, you can use the following technique to specify a parameter string of up to 300 characters.

```
// EXEC [PGM=]program_name,                                          X
               PARM='parameter_string_segment',                      X
               PARM='parameter_string_segment',                      X
               PARM='parameter_string_segment'
```

Using this technique, up to three instances of the PARM parameter can be specified on an EXEC statement, and each *parameter_string_segment* can be up to 100

characters in length. The following example shows how you might code an EXEC statement and pass a 140-character string consisting of run-time options and program arguments.

```
// EXEC TESTPGM,PARM='ABTERMENC(ABEND) ALL31(ON) NATLANG(ENU) RPTOPTS(OX
               N) RPTSTG(ON) TERMTHDACT(MSG)/RUNDATE=19961213 DBNA',   X
               PARM='ME=ODBMST TRANFLE=OTRNFLE RPTFLE=ORPTFLE'
```

Using the iconv Utility for C

The `iconv` utility uses the `iconv_open()`, `iconv()`, and `iconv_close()` functions to convert the input file records from the coded character set definition for the input code page to the coded character set definition for the output code page. There is one record in the output file for each record in the input file. No padding or truncation of records is performed. For information on the `iconv` utility, see *LE/VSE C Run-Time Programming Guide*.

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

Using the genxlt Utility for C

The `genxlt` utility reads character conversion information from an input file and generates an object module containing a conversion table. The input file contains directives that are acted upon by the `genxlt` utility to produce the compiled version of the conversion table. The object module, when link-edited, is used by the `iconv` utility and `iconv` functions. For more information on the `genxlt` utility, see *LE/VSE C Run-Time Programming Guide*.

Chapter 5. Using Run-Time Options

This chapter shows you how to specify run-time options as installation defaults, application defaults, in JCL, in assembler exits, or in your source code.

Understanding the Basics

LE/VSE provides run-time options with which you can control certain aspects of your program's processing. You can set the default values for most of these options at installation time. Table 12 lists the LE/VSE run-time options and gives a brief description of each. For more detailed information on the syntax and use of LE/VSE run-time options, and how LE/VSE run-time options map to specific HLL options, see *LE/VSE Programming Reference*.

Table 12. Summary of LE/VSE Run-Time Options

Run-Time Option	Description
ABPERC	Specifies a VSE cancel code, a program-interruption code, or a user cancel code to be exempted from LE/VSE condition handling.
ABTERMENC	Sets the enclave termination behavior for an enclave ending with an unhandled condition of severity 2 or greater.
AIXBLD|NOAIXBLD	(COBOL only) Invokes the access method services (AMS) for VSAM indexed and relative data sets to complete the file and index definition procedures for COBOL routines.
ALL31	Specifies whether an application can run entirely in AMODE(31), or whether the application has one or more AMODE(24) routines.
ANYHEAP	Controls the allocation of library heap storage that is not restricted to a location below 16MB.
ARGPARSE|NOARGPARSE	(C only) Specifies whether arguments on the *command line* are to be parsed in the usual C format. This option is restricted to applications in which C is the main routine. You can only specify it using the C `#pragma runopts` directive.
BELOWHEAP	Controls the allocation of library heap storage that must be located below 16MB. The heap controlled by BELOWHEAP is intended for items such as control blocks used for I/O.
CBLOPTS	(COBOL only) Specifies the format of the parameter string on application invocation when the main routine is COBOL. CBLOPTS determines whether run-time options or program arguments appear first in the parameter string. You can only specify CBLOPTS in CEEDOPT or CEEUOPT.
CBLPSHPOP	(COBOL only) Determines whether CICS PUSH HANDLE and CICS POP HANDLE commands are issued when a COBOL subroutine is called.
CHECK	(COBOL only) Activates error checking within an application. Index, subscript, and reference modification errors are checked.
COUNTRY	Sets the default formats for date, time, currency symbol, decimal separator, and thousands separator, based on a specified country.

© Copyright IBM Corp. 1991, 2005

Table 12. Summary of LE/VSE Run-Time Options (continued)

Run-Time Option	Description
DEBUG|NODEBUG	(COBOL only) Activates the COBOL batch debugging language specified by the USE FOR DEBUGGING declarative.
DEPTHCONDLMT	Specifies the extent to which conditions can be nested.
ENV	(C only) Specifies the operating environment for your C application. It is only required for running with DL/I, and is restricted to applications in which C is the main routine. You can only specify ENV using the C `#pragma runopts` directive.
ENVAR	(C only) Sets initial values for specified environment variables. This option is restricted to applications in which C is the main routine.
ERRCOUNT	Specifies how many conditions of severity 2, 3, and 4 can occur before an enclave terminates abnormally.
EXECOPS|NOEXECOPS	(C only) Specifies whether you can enter run-time options on the command line. This option is restricted to applications in which C is the main routine. C applications can use the `#pragma runopts` directive to specify these options or use the EXECOPS/NOEXECOPS compile-time options.
HEAP	Controls the allocation of the initial heap and of additional heaps created with the CEECRHP callable service, and specifies how that storage is managed.
HEAPCHK	Provides a checking facility to verify that the heap storage has not been damaged.
LIBSTACK	Controls the allocation of a thread's library stack storage. This stack is used by LE/VSE and HLL library routines that require save areas below 16MB.
MSGFILE	Specifies the filename of the file where all run-time diagnostics and reports generated by the RPTOPTS and RPTSTG run-time options are written.
MSGQ	Specifies the number of instance-specific information (ISI) blocks LE/VSE allocates on a per-thread basis for use by the application.
NATLANG	Specifies the initial national language to be used for the run-time environment, including error messages, month names, and day-of-the-week names.
PLIST	(C only) Specifies the format of the parameters received by a C application on invocation. This option is restricted to applications where C is the main routine. You can only specify it using the C `#pragma runopts` directive.
REDIR|NOREDIR	(C only) Specifies whether you can enter directions for `stdin`, `stderr`, and `stdout` from the command line. This option is restricted to applications in which C is the main routine. You can only specify it using the C `#pragma runopts` directive.
RETZERO	(COBOL only) Controls the value of the COBOL RETURN-CODE special register at run-unit termination.
RPTOPTS	Generates a report of the run-time options in effect while an application was running.

Table 12. Summary of LE/VSE Run-Time Options (continued)

Run-Time Option	Description
RPTSTG	Generates a report of the storage used by an application.
RTEREUS	This option is provided for compatibility with the VS COBOL II RTEREUS option; however, its use is not recommended due to restrictions with multienclave or multilanguage applications. Although RTEREUS can be specified in CEEUOPT or CEEDOPT, it is not recommended as an installation default.
STACK	Controls the allocation of a thread's stack storage.
STORAGE	Controls the initial content of storage when allocated and freed, and the amount of storage that is reserved for the out-of-storage condition.
TERMTHDACT	Sets the level of information that is produced when LE/VSE percolates a condition of severity 2 or greater beyond the first routine's stack frame. Also specifies the dump output destination for the CICS environment.
TEST | NOTEST	Specifies the conditions under which a debug tool assumes control when the user application is being initialized.
TRACE	Activates LE/VSE run-time library tracing and controls the size of the trace table, the type of trace, and whether the trace table should be dumped unconditionally upon termination of the application.
TRAP	Specifies how LE/VSE routines handle abends and program interrupts. LE/VSE expects TRAP(ON,MIN) to be in effect for successful execution of the application.
UPSI	(COBOL only) Sets the eight UPSI switches on or off for applications that use COBOL programs.
USRHDLR | NOUSRHDLR	Specifies the name of a user condition handler, if any, to be registered at stack frame 0.
XUFLOW	Specifies whether an exponent underflow causes a program interrupt.

Specifying Run-time Options

You can specify LE/VSE run-time options in the following ways:

As installation defaults

The CEEDOPT assembler language source file establishes installation defaults (for the batch environment) using the CEEXOPT macro. The file initially contains IBM-supplied default values for each of the LE/VSE run-time options. The syntax for CEEXOPT is presented in the section, "CEEXOPT Invocation Syntax" on page 40. During installation of LE/VSE, the default values contained in the CEEDOPT source file can be edited and assembled to create the CEEDOPT object module. All batch applications that run in the common run-time environment operate using these default values for the run-time options. If LE/VSE is installed in the default sublibraries, the CEEDOPT source file and object module reside in the PRD2.SCEEBASE sublibrary.

It is possible to associate a non-overridable attribute with each individual run-time option. Each option in CEEDOPT must be specified as either

overridable (OVR) or non-overridable (NONOVR). This allows the installation to enforce options that are critical to the overall LE/VSE operating environment.

LE/VSE also provides the CEECOPT source program to establish installation defaults for run-time options under CICS. If LE/VSE is installed in the default sublibraries, the CEECOPT source file, and the CEECOPT object module it generates, reside in the PRD2.SCEEBASE sublibrary.

Both the CEEDOPT and CEECOPT object modules contain CEEDOPT CSECT.

For more information on specifying options at installation, see *LE/VSE Customization Guide*.

In the assembler user exit

With the assembler user exit you can override all other sources of run-time options except those specified as non-overridable in the installation defaults. See "CEEBXITA Assembler User Exit Interface" on page 323 for information about how to specify a list of run-time options in the assembler user exit.

As application defaults

The CEEUOPT assembler language source program sets application defaults using the CEEXOPT macro. Like CEEDOPT, CEEUOPT can be edited and assembled to create an object module, CEEUOPT, that can be linked with an application.

As noted above, CEEDOPT establishes installation defaults using the CEEXOPT macro. When the program runs, the options specified in CEEUOPT override any corresponding overridable CEEDOPT options.

CEEUOPT must be linked with the main program of your application in order to establish application defaults.

In JCL

You can specify run-time options in the PARM parameter of the JCL EXEC statement. These run-time options override all other sources of run-time options except those provided by the assembler user exit, and those specified as non-overridable in the installation defaults. See "Specifying Run-Time Options in the EXEC Statement" on page 30 for details.

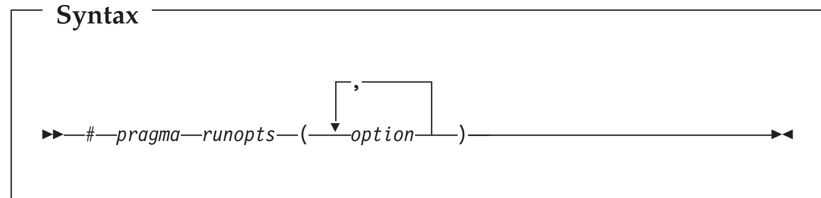
Note: If you use LE/VSE preinitialization services to create and initialize the common run-time environment, LE/VSE does not honor run-time options specified in the PARM parameter of your JCL EXEC statement. For information about how to specify run-time options when using LE/VSE preinitialization services, see "Specifying Run-Time Options and Program Arguments" on page 367.

In your source code

C C provides the `#pragma runopts` directive, with which you can specify run-time options in your source code.

You must specify `#pragma runopts` in the source file that contains your main function, before the first C statement. Only comments and other pragmas can precede `#pragma runopts`.

Specify `#pragma runopts` as follows:



option is an LE/VSE run-time option.

For more information about using C pragmas, see *LE/VSE C Run-Time Programming Guide*.

PL/I Run-time options can be specified in a PL/I source application by means of the following declaration:

```
DCL PLIXOPT CHAR(length) VAR INIT('string')
           STATIC EXTERNAL;
```

where *string* is a list of options separated by commas or blanks, and *length* is a constant equal to or greater than the length of *string*. Run-time options in PLIXOPT are parsed by the compiler. The PL/I VSE compiler produces the CEEUOPT CSECT for the PLIXOPT string.

If more than one external procedure in a job declares PLIXOPT as STATIC EXTERNAL, only the first link-edited string is available at run time.

Each time a PL/I application runs, the default run-time options established at installation time apply unless overridden by a PLIXOPT string in the source program or in the PARM parameter of the JCL EXEC statement. Options specified in the PARM parameter override those specified in the PLIXOPT string.

Order of Precedence

It is possible for all the methods listed above to be used for a given application. The order of precedence (from highest to lowest) between option specification methods is:

1. Options defined at installation time that have the non-overridable (NONOVR) attribute.
2. Options specified by the assembler user exit.
3. Options specified on invocation of the application. Under CICS, options set using the CLER transaction.
4. Options specified within the source program, or options specified in CEEUOPT and link-edited with the application.

   If you use both these methods to specify run-time options, the precedence is determined by the order the object modules are included when you link edit your application. If you include the object module for the source program first, the options specified within the source program will be accepted and the options specified in CEEUOPT will be ignored. If you include the CEEUOPT object module first, the options specified in CEEUOPT will be accepted and the options specified within the source program will be ignored. You can use linkage editor control statements to control the order in which the object

modules are included. (Note that the z/VSE linkage editor produces an information-only message when a duplicate CEEDOPT CSECT is detected, and ignores the second CSECT.)

5. Option defaults defined at installation time.

Specifying Suboptions in Run-Time Options

Use commas to separate suboptions of run-time options. If you do not specify a suboption, you must still specify the comma to indicate its omission, for example `STACK(,,ANYWHERE,FREE)`. However, trailing commas are not required; `STACK(4K,4K,ANYWHERE)` is valid. If you do not specify any suboptions, either of the following is valid: `STACK`, or `STACK()`.

Specifying Run-Time Options and Program Arguments

In order to distinguish run-time options from program arguments that are passed to LE/VSE, the options and program arguments are separated by a slash (/). (For more information on program arguments, see “Argument Lists and Parameter Lists” on page 50.)

Run-time options usually precede program arguments whenever they are specified in JCL. The possible combinations are described in Table 13. You can override this format to ensure compatibility with COBOL applications. See “COBOL Compatibility Considerations” on page 39 for more information.

Table 13. Formats for Specifying Run-Time Options and Program Arguments

Possible combinations	Format
Only run-time options are present	run-time options/
Only program arguments are present • If a slash is present in the arguments, a preceding slash is mandatory. • If a slash is NOT present in the arguments, a preceding slash is optional.	 /program arguments program arguments or /program arguments
Both run-time options and program arguments are present	run-time options/program arguments

You can use the callable service CEE5PRM to retrieve program arguments. For information on CEE5PRM, see *LE/VSE Programming Reference*.

In the following example, an object module called `MYPROG` is link-edited and run. The code in the example overrides the LE/VSE defaults for the RPTOPTS and MSGFILE run-time options.

```
// JOB     jobname
// DLBL    IJSYSLN,'%LEVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,       X
               RECORDS=(400,600)
// DLBL    OPTRPRT,'fileid',0,SD
// EXTENT  SYSnnn,volser,1,0,start,tracks
// ASSGN   SYSnnn,DISK,VOL=volser,SHR
// LIBDEF  OBJ,SEARCH=(userlib.sublib,lelib.sublib)
// LIBDEF  PHASE,SEARCH=(lelib.sublib)
// OPTION  LINK
   INCLUDE MYPROG
// EXEC    LNKEDT
// EXEC    ,PARM='RPTOPTS(ON),MSGFILE(OPTRPRT)/'
/*
/&
```

C Compatibility Considerations

C provides the `#pragma runopts` directive for you to specify run-time options in your source code. If the main routine is C and `#pragma runopts(execops)` is in effect (the default), you can pass run-time options in the PARM parameter of the JCL EXEC statement. Run-time options must be followed by a slash (/).

If the main routine is C and `#pragma runopts(noexecops)` is specified in the source, you cannot enter run-time options in the PARM parameter. LE/VSE interprets the entire string in the PARM parameter, including run-time options if present, as program arguments to the main routine.

See *LE/VSE Programming Reference* for a description of the EXECOPS run-time option.

COBOL Compatibility Considerations

VS COBOL II supports an order of run-time options and program arguments that is the reverse of that expected by LE/VSE. In VS COBOL II, the specification order is program arguments/run-time options.

To ensure compatibility with VS COBOL II, LE/VSE provides the run-time option CBLOPTS, which allows you to choose whether run-time options or program arguments are expected first in the parameter list. CBLOPTS can only be specified in the user options CSECT, CEEUOPT, or in the installation default run-time options CSECT, CEEDOPT. You can specify a slash (/) as part of the program arguments with CBLOPTS(ON) or CBLOPTS(OFF).

CBLOPTS(ON) allows the existing VS COBOL II format of the invocation character string to continue working (program arguments followed by run-time options). When CBLOPTS(ON) is specified, the last slash in a string delineates the program arguments from the run-time options. Anything before the last slash is interpreted as a program argument. Conversely, when CBLOPTS(OFF) is specified, the first slash delineates the run-time options from the program arguments. Anything after the first slash is interpreted as a program argument.

CBLOPTS is honored **only** when a COBOL routine is the main routine in the application. See *LE/VSE Programming Reference* for more information.

PL/I Compatibility Considerations

If a PL/I main program is compiled with the NOEXECOPS procedure option, run-time options cannot be specified in the PARM parameter of the JCL EXEC statement. If run-time options are specified, they are passed as program arguments. The effect of NOEXECOPS is described in Appendix B, "Using Operating System and Subsystem Parameter List Formats," on page 403.

CEEXOPT Invocation Syntax

Use the CEEXOPT macro to establish installation and programmer default options.

- When invoked during the assembly of the CEEDOPT source program at installation time, CEEXOPT creates the CEEDOPT object module, which establishes batch installation default options. LE/VSE run-time options (except those that are C specific) must be specified in CEEDOPT. Each option in CEEDOPT must be designated as either overridable (OVR) or non-overridable (NONOVR). In addition, a valid value must be specified for each suboption of each run-time option.
- When invoked during the assembly of the CEECOPT source program at installation time, CEEXOPT creates the CEECOPT object module, which establishes CICS installation default options. LE/VSE run-time options (except those that are C specific) must be specified in CEECOPT. Each option in CEECOPT must be designated as either overridable (OVR) or non-overridable (NONOVR). In addition, a valid value must be specified for each suboption of each run-time option.
- The CEEXOPT macro also creates the CEEUOPT object module when CEEUOPT is assembled. CEEUOPT can be linked with an application program to establish user default options. Options in CEEUOPT's invocation of CEEXOPT must **not** be designated as overridable or non-overridable. However, their suboption values take precedence over those of any corresponding overridable CEEDOPT option values.

To invoke CEEXOPT, use the format of the IBM-supplied templates CEEDOPT, CEECOPT, and CEEUOPT, as shown in Figure 14 on page 41, Figure 15 on page 42, and Figure 16 on page 43, respectively.

Figure 14 shows a sample of the IBM-supplied version of CEEDOPT with the default suboption values for each of the options, which establish installation defaults for batch.

```
CEEDOPT  CSECT
CEEDOPT  AMODE ANY
CEEDOPT  RMODE ANY
         CEEXOPT ABPERC=((NONE),OVR),                                  X
               ABTERMENC=((ABEND),OVR),                                X
               AIXBLD=((OFF),OVR),                                     X
               ALL31=((OFF),OVR),                                      X
               ANYHEAP=((16K,8K,ANYWHERE,FREE),OVR),                   X
               BELOWHEAP=((8K,4K,FREE),OVR),                           X
               CBLOPTS=((ON),OVR),                                     X
               CBLPSHPOP=((OFF),OVR),                                  X
               CHECK=((OFF),OVR),                                      X
               COUNTRY=((US),OVR),                                     X
               DEBUG=((OFF),OVR),                                      X
               DEPTHCONDLMT=((10),OVR),                                X
               ENVAR=((''),OVR),                                       X
               ERRCOUNT=((20),OVR),                                    X
               HEAP=((32K,32K,ANYWHERE,KEEP,8K,4K),OVR),               X
               HEAPCHK=((OFF,1,0),OVR),                                X
               LIBSTACK=((12K,4K,FREE),OVR),                           X
               MSGFILE=((SYSLST),OVR),                                 X
               MSGQ=((15),OVR),                                        X
               NATLANG=((UEN),OVR),                                    X
               NOTEST=((ALL,*,PROMPT,''),OVR),                         X
               NOUSRHDLR=((),OVR),                                     X
               RETZERO=((OFF),OVR),                                    X
               RPTOPTS=((OFF),OVR),                                    X
               RPTSTG=((OFF),OVR),                                     X
               RTEREUS=((OFF),OVR),                                    X
               STACK=((128K,128K,BELOW,KEEP),OVR),                     X
               STORAGE=((00,NONE,NONE,32K),OVR),                       X
               TERMTHDACT=((TRACE,,0),OVR),                            X
               TRACE=((OFF,4K,DUMP,LE=0),OVR),                         X
               TRAP=((ON,MAX),OVR),                                    X
               UPSI=((00000000),OVR),                                  X
               XUFLOW=((AUTO),OVR)
*/*                                                                  */
*/* The below macro requires valid VSE/POWER settings for each of    */
*/* the options. If this is not done, failures may result and lost   */
*/* output.                                                          */
*/* The options NODE and USERID are optional. However, if a NODE is  */
*/* specified, then a valid USERID MUST ALSO be specified. If you    */
*/* require the behaviour of the * in the node parameter, omit the   */
*/* node setting and just supply a USERID setting. Specifying an *   */
*/* in the NODE parameter is NOT VALID.                              */
*/* To get a report of the current LSTQ options settings, set        */
*/* RPTOPTS(ON) via a support method and the resulting report will   */
*/* include a LSTQ options report.                                   */
CEELSTQ  CEELOPT CLASS=L,                                              X
               DISP=D,                                                 X
               NODE=,                                                  X
               USERID=
         DC    C'5686-CF7-32-81K (C) COPYRIGHT IBM CORP. 1991, 2004.'
         DC    C'LICENSED MATERIALS - PROPERTY OF IBM'
         END
```

Figure 14. IBM-Supplied Batch Installation Default Options Source Program, CEEDOPT

Figure 15 shows a sample of the IBM-supplied version of CEECOPT with the default sub-option values for each of the options, which establish installation defaults for CICS.

```
CEEDOPT  CSECT
CEEDOPT  AMODE ANY
CEEDOPT  RMODE ANY
         CEEXOPT ABPERC=((NONE),OVR),                                  X
               ABTERMENC=((ABEND),OVR),                                X
               AIXBLD=((OFF),OVR),                                     X
               ALL31=((ON),OVR),                                       X
               ANYHEAP=((4K,4080,ANYWHERE,FREE),OVR),                  X
               BELOWHEAP=((4K,4080,FREE),OVR),                         X
               CBLOPTS=((ON),OVR),                                     X
               CBLPSHPOP=((ON),OVR),                                   X
               CHECK=((OFF),OVR),                                      X
               COUNTRY=((US),OVR),                                     X
               DEBUG=((OFF),OVR),                                      X
               DEPTHCONDLMT=((10),OVR),                                X
               ENVAR=(('')，OVR),                                      X
               ERRCOUNT=((20),OVR),                                    X
               HEAP=((4K,4080,ANYWHERE,KEEP,4K,4080),OVR),             X
               HEAPCHK=((OFF,1,0),OVR),                                X
               LIBSTACK=((4K,4080,FREE),OVR),                          X
               MSGFILE=((CESE),OVR),                                   X
               MSGQ=((15),OVR),                                        X
               NATLANG=((UEN),OVR),                                    X
               NOTEST=((ALL,*,PROMPT,''),OVR),                         X
               NOUSRHDLR=((),OVR),                                     X
               RETZERO=((OFF),OVR),                                    X
               RPTOPTS=((OFF),OVR),                                    X
               RPTSTG=((OFF),OVR),                                     X
               RTEREUS=((OFF),OVR),                                    X
               STACK=((4K,4080,ANYWHERE,KEEP),OVR),                    X
               STORAGE=((00,NONE,NONE,0K),OVR),                        X
               TERMTHDACT=((TRACE,MSGFL,0),OVR),                       X
               TRACE=((OFF,4K,DUMP,LE=0),OVR),                         X
               TRAP=((ON,MAX),OVR),                                    X
               UPSI=((00000000),OVR),                                  X
               XUFLOW=((AUTO),OVR)
*        DC    C'5686-066-32-65K (C) COPYRIGHT IBM CORP. 1991, 2001.'
*        DC    C'LICENSED MATERIALS - PROPERTY OF IBM'
*/*                                                                 */
*/* The below macro requires valid VSE/POWER settings for each of    */
*/* the options. If this is not done, failures will result with the  */
*/* CEEL011S message being displayed. The information displayed in   */
*/* this message can be used to determine the failure by referencing */
*/* the VSE/POWER Application Programming Guide with the displayed   */
*/* VSE/POWER return code and feedback code.                         */
*/* The options NODE and USERID are optional. However, if a NODE is  */
*/* specified, then a valid USERID MUST ALSO be specified. If you    */
*/* require the behaviour of the * in the node parameter, omit the   */
*/* node setting and just supply a USERID setting. Specifying an *   */
*/* in the NODE parameter is NOT VALID.                              */
*/* To get a report of the current LSTQ options settings, run the    */
*/* supplied NEWC (or your locally defined version) CICS transaction.*/
```

Figure 15. IBM-Supplied CICS Installation Default Options Source Program, CEECOPT (Part 1 of 2)

```
*/* This will produce a LSTQ options report and a LE/VSE run-time    */
*/* options report. This transaction will also reload the LSTQ and   */
*/* run-time options dynamically while CICS is still active.         */
CEELSTQ  CEELOPT CLASS=L,                                              X
               DISP=D,                                                 X
               NODE=,                                                  X
               USERID=
         DC    C'5686-CF7-32-81K (C) COPYRIGHT IBM CORP. 1991, 2004.'
         DC    C'LICENSED MATERIALS - PROPERTY OF IBM'
         END
```

Figure 15. IBM-Supplied CICS Installation Default Options Source Program, CEECOPT (Part 2 of 2)

Figure 16 shows a sample of the IBM-supplied version of CEEUOPT with the default suboption values for each of the options, which establish application defaults.

```
CEEUOPT  CSECT
CEEUOPT  AMODE ANY
CEEUOPT  RMODE ANY
         CEEXOPT ABPERC=(NONE),                                        X
               ABTERMENC=(ABEND),                                 @01C X
               AIXBLD=(OFF),                                           X
               ALL31=(OFF),                                            X
               ANYHEAP=(16K,8K,ANYWHERE,FREE),                         X
               BELOWHEAP=(8K,4K,FREE),                                 X
               CBLOPTS=(ON),                                           X
               CBLPSHPOP=(OFF),                                   @01C X
               CHECK=(OFF),                                            X
               COUNTRY=(US),                                           X
               DEBUG=(OFF),                                            X
               DEPTHCONDLMT=(10),                                      X
               ENVAR=(''),                                             X
               ERRCOUNT=(20),                                          X
               HEAP=(32K,32K,ANYWHERE,KEEP,8K,4K),                     X
               HEAPCHK=(OFF,1,0),                                      X
               LIBSTACK=(8K,4K,FREE),                                  X
               MSGFILE=(SYSLST),                                       X
               MSGQ=(15),                                              X
               NATLANG=(UEN),                                          X
               NOTEST=(ALL,*,PROMPT,''),                               X
               NOUSRHDLR=(),                                           X
               RETZERO=(OFF),                                          X
               RPTOPTS=(OFF),                                          X
               RPTSTG=(OFF),                                           X
               RTEREUS=(OFF),                                          X
               STACK=(128K,128K,BELOW,KEEP),                           X
               STORAGE=(00,NONE,NONE,32K),                        @01C X
               TERMTHDACT=(TRACE,,0),                             @02C X
               TRACE=(OFF,4K,DUMP,LE=0),                               X
               TRAP=(ON,MAX),                                     @01C X
               UPSI=(00000000),                                        X
               XUFLOW=(AUTO)
         DC    C'5686-CF7-32-75K (C) COPYRIGHT IBM CORP. 1991, 2002.'
         DC    C'LICENSED MATERIALS - PROPERTY OF IBM'
         END
```

Figure 16. IBM-Supplied Application Default Options Source Program, CEEUOPT

Notes on CEEXOPT Invocation

You should be aware of the following considerations when invoking CEEXOPT:

- A continuation character (X in the source) must be present in column 72 on each line of the CEEXOPT invocation except the last line. This applies to both CEEUOPT and CEEDOPT.
- Options and suboptions must be specified in upper case. Only suboptions that are strings can be specified in mixed case or lowercase. For example, both MSGFILE=(SYSLST) and MSGFILE=(syslst) are acceptable. ALL31=(off) is not acceptable.
- A comma must end each option except for the final option. If the comma is omitted, everything following the option is treated as a comment.
- If one of the string suboptions contains a special character, such as embedded blank or unmatched right or left parenthesis, the string must be enclosed in single apostrophes (' '), not in double quotation marks (" "). (A null string can be specified with either adjacent single apostrophes or adjacent double quotation marks.)

  To obtain a single apostrophe (') or a single ampersand (&) within a string, two instances of the character must be specified. The pair is counted as only one character in determining whether the maximum allowable string length has been exceeded, and in setting the effective length of the string.
- Macro instruction operands cannot exceed 255 characters in length. Therefore, it is not possible for each suboption of the TEST|NOTEST option to attain the maximum allowable length normally permitted by LE/VSE. For example, the *command* suboption of the TEST option permits 250 characters, while the *preference_file* suboption allows 80. The total number of characters for these two suboptions, therefore, exceeds that allowed by the CEEXOPT macro. See *LE/VSE Programming Reference* for further information. If the number of characters to the right of the equal sign is greater than 255 for any keyword parameter in the CEEXOPT invocation in CEEUOPT or CEEDOPT, a return code of 12 is produced for the assembly, and no options are parsed.
- Avoid unmatched apostrophes in any string. The error cannot be captured within CEEXOPT itself; instead, the assembler produces a message such as

  ```
  ASMA063  *** ERROR ***  NO ENDING APOSTROPHE
  ```

  which bears no particular relationship to the suboption in which the apostrophe was omitted. Furthermore, none of the options is properly parsed if this mistake is made.
- You can completely omit the specification of any option in CEEUOPT. Default values are then supplied for each of the missing options.

  In CEEUOPT, IBM recommends that you omit any options you do not wish to change. The options you omit from the macro will default to the installation-wide defaults you set in CEEDOPT or CEECOPT.
- In CEEUOPT, you can use commas to indicate the omission of one or more suboptions for options having more than one suboption. For example, if you wish to specify only the second suboption of the STORAGE option, the omission of the 1st, 3rd, and 4th suboptions can be indicated in any of the following ways:

  ```
  STORAGE=(,NONE),                                                       X
  STORAGE=(,NONE,),                                                      X
  STORAGE=(,NONE,,),                                                     X
  ```

  Because suboptions are positional parameters, do not omit the comma if the corresponding suboption is omitted and another suboption follows.

Note: If you specify an option in the CEEUOPT, there are special defaults for omitted sub-options. You can find these default values under the "Usage Notes" heading for the related run-time options.

- Options that allow only one suboption do not need to enclose that suboption in parentheses. For example, the COUNTRY option can be specified in CEEUOPT in either of the following ways:

```
COUNTRY=(US),                                                        X
COUNTRY=US,                                                          X
```

Performance Considerations

In CEEUOPT, code only those options that you want to change. This enhances performance by minimizing the number of options lines LE/VSE must scan. Options that are to remain the same as the installation defaults do not need to be repeated. For example, if the only change you want to make is to define STACK with an initial value of 64K and an increment of 64K, include the following in CEEUOPT:

```
CEEUOPT CSECT
CEEUOPT AMODE ANY
CEEUOPT RMODE ANY
        CEEXOPT STACK=(64K,64K,BELOW,KEEP)
END
```

WARNING: If you plan to use application-specific options with a CICS application, you should review the IBM-supplied values, and ensure they are appropriate for your CICS application. For values that are applicable to CICS, see the storage report produced when using RPTSTG(ON) or the supplied CEECOPT member (shown in Figure 15 on page 42). Otherwise, your CICS system might suffer from performance degradation and/or storage problems.

Printing CICS-Wide Run-Time Options to Console

This function allows you to print LE/VSE CICS-wide run-time options to your z/VSE console. This provides an alternative to the global CICS-wide setting that is possible using the LE/VSE run-time option `RPTOPTS(ON)`. This function allows you to avoid producing large output on the LE/VSE destination `CESE` (in contrast to the `RPTOPTS(ON)` run-time option which might fill up this queue, if defined as a file, and generate message CEE3492S).

From VSE/ESA 2.5 onwards, LE/VSE program EDCYCROP is shipped with the CICS transid already set to 'ROPC'. Transaction ROPC is already BSM security-enabled, and can be used immediately. When this transaction has been invoked on a CICS terminal, LE/CICS-wide default run-time options (CEECOPT) appear on the console.

Notes:

1. If you have set the ENVAR run-time option of the CICS-wide Assember User Exit, you can list this option in the CICS-Wide Options report sent to your z/VSE console. In the example report shown in Figure 17 on page 46, the ENVAR run-time option has been included.
2. The function is available with both CICS/VSE 2.3 and CICS Transaction Server.
3. For VSE systems before VSE/ESA 2.5, you can specify your own CICS transid for LE/VSE program EDCYCROP.

```
F2 0114 Options Report for Enclave EDCYCROP 04/27/04 9:13:46 AM
F2 0114 Language Environment for VSE/ESA V1 R4.4
F2 0114
F2 0114 LAST WHERE SET          OPTION
F2 0114 -----------------------------------------------------------------------------
F2 0114 Installation default      ABPERC(NONE)
F2 0114 Installation default      ABTERMENC(ABEND)
F2 0114 Installation default    NOAIXBLD
F2 0114 Installation default      ALL31(ON)
F2 0114 Installation default      ANYHEAP(4096,4080,ANYWHERE,FREE)
F2 0114 Installation default      BELOWHEAP(4096,4080,FREE)
F2 0114 Installation default      CBLOPTS(ON)
F2 0114 Installation default      CBLPSHPOP(ON)
F2 0114 Installation default      CHECK(OFF)
F2 0114 Installation default      COUNTRY(US)
F2 0114 Installation default    NODEBUG
F2 0114 Installation default      DEPTHCONDLMT(10)
F2 0114 Installation default      ENVAR("")
F2 0114 Installation default      ERRCOUNT(20)
F2 0114 Installation default      HEAP(4096,4080,ANYWHERE,KEEP,4096,4080)
F2 0114 Installation default      HEAPCHK(OFF,1,0)
F2 0114 Installation default      LIBSTACK(4096,4080,FREE)
F2 0114 Installation default      MSGFILE(CESE)
F2 0114 Installation default      MSGQ(15)
F2 0114 Installation default      NATLANG(UEN)
F2 0114 Installation default      RETZERO(OFF)
F2 0114 Installation default      RPTOPTS(OFF)
F2 0114 Installation default      RPTSTG(OFF)
F2 0114 Installation default    NORTEREUS
F2 0114 Installation default      STACK(4096,4080,ANYWHERE,KEEP)
F2 0114 Installation default      STORAGE(00,NONE,NONE,0)
F2 0114 Installation default      TERMTHDACT(TRACE,MSGFL,0)
F2 0114 Installation default    NOTEST(ALL,"*","PROMPT","")
F2 0114 Installation default      TRACE(OFF,4096,DUMP,LE=0)
F2 0114 Installation default      TRAP(ON,MAX)
F2 0114 Installation default      UPSI(00000000)
F2 0114 Installation default    NOUSRHDLR()
F2 0114 Installation default      XUFLOW(AUTO)
```

Figure 17. Sample of LE/CICS-Wide Options Printed to Console

Part 2. Preparing an Application to Run with LE/VSE

Chapter 6. Using LE/VSE Parameter List Formats 49
Understanding the Basics 49
Argument Lists and Parameter Lists 50
Passing Arguments between Routines 50
Preparing Your Main Routine to Receive Parameters 51
PL/I Argument-Passing Considerations 55

Chapter 7. Routines That Must Be Reentrant . . 57
Understanding the Basics 57
Making Your C Program Reentrant 57
Natural Reentrancy 57
Constructed Reentrancy 57
Generating a Reentrant C Object Module . . . 58
Making Your COBOL Routine Reentrant 58
Making Your PL/I Routine Reentrant 58
Installing a Reentrant Phase 58

Running an application is generally the same under LE/VSE as in earlier versions of a language's run-time. However, in order to take advantage of some of the features that a common execution environment offers, you must consider a number of different things when preparing an application to run in LE/VSE.

When running applications in LE/VSE, you must consider the target operating environment. Currently, under VSE and CICS, the way that parameters are passed differs. To ensure consistency, LE/VSE standardizes the parameters as much as possible. It is therefore important for you to know what LE/VSE does to the format to ensure this consistency. Parameter list format information is detailed in Chapter 6, "Using LE/VSE Parameter List Formats," on page 49 and Appendix B, "Using Operating System and Subsystem Parameter List Formats," on page 403.

In addition to describing parameter list formats, this part describes how to manage return codes, and offers suggestions on how to make your LE/VSE-conforming applications reentrant.

© Copyright IBM Corp. 1991, 2005

Chapter 6. Using LE/VSE Parameter List Formats

This chapter describes how to pass parameters to external routines under LE/VSE. The methods described do not apply to internal routines or to compiled code that invokes its own library routines. Each LE/VSE-conforming HLL might have its own method for transferring control and passing arguments between internal routines.

Understanding the Basics

When writing an LE/VSE-conforming application, it is important to consider how parameters are passed to the application on invocation. The type of parameter list passed to LE/VSE when an application is run varies according to whether you are running in the batch environment or under CICS. In the batch environment, LE/VSE repackages the parameter string specified in the EXEC statement so that what is actually passed to the main routine is a halfword-prefixed character string. If you set up your C, COBOL, or PL/I main routine according to the rules of the language, you generally do not need to do anything special to receive parameters from the operating system.

Under CICS or DL/I, however, the parameter format that is passed might be different from what your main routine expects. In these cases, you must explicitly code your main routine to accept the format of the parameters as they are passed by CICS or DL/I.

"Preparing Your Main Routine to Receive Parameters" on page 51 contains examples of how to code your main routine to receive parameters under any supported operating system or subsystem.

Additionally, some HLLs such as C and PL/I provide options that enable you to specify the format of the parameter list you expect to be passed to your main routine. For example, C programmers can specify the PLIST run-time option, which determines the parameter list format. Refer to one of the following for information on which settings you should select to run an application:

For this type of application	Refer to
Batch without DL/I	Table 16 on page 52
Batch with DL/I	Table 17 on page 53
CICS	Table 18 on page 54
Assembler calling HLL	Table 19 on page 54

When running most main routines, you do not need to explicitly access the parameter list. In some cases, you might want to inquire about the parameters passed to your main routine; LE/VSE provides the CEE5PRM callable service to do this (see *LE/VSE Programming Reference*). In addition, C allows you to investigate passed parameters using constructs within the HLL itself. For more information, see "C Parameter Passing Considerations" on page 403.

© Copyright IBM Corp. 1991, 2005

Argument Lists and Parameter Lists

The terminology used to describe passing parameters to and from routines currently differs among LE/VSE-conforming HLLs. Figure 18 summarizes the terminology used with LE/VSE. In Figure 18, a calling routine passes an *argument* list to a called routine. That same list is referred to as a *parameter* list when it is received by the called routine. Under LE/VSE, the formats of the argument and parameter lists are identical. The only difference between the two terms is whether it is being used from the point of view of the calling or the called routine.

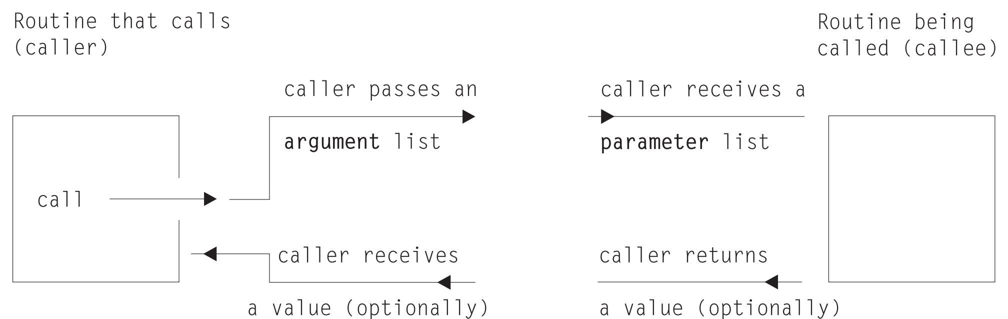


Figure 18. Call Terminology Refresher

Passing Arguments between Routines

LE/VSE-conforming HLLs use the semantic terms *by value* and *by reference* for passing arguments:

By value
: Only the value of the object is passed. Any changes made by the called routine to the argument value are not reflected in the calling routine.

By reference
: The address of the object is passed. Changes made by the called routine to the argument value are reflected in the calling routine.

Under LE/VSE you can pass arguments directly and indirectly as follows:

Direct The value of the argument is placed directly in the argument list body. You cannot pass an argument by reference (direct).

Indirect
: The body of the argument list contains a pointer to the argument value.

Table 14 summarizes the semantic terms by value and by reference and the direct and indirect methods for passing arguments. The table shows what is passed to routines.

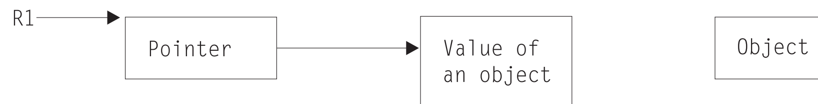
Table 14. Semantic Terms and Methods for Passing Arguments in LE/VSE

	By Value	**By Reference**
Direct	The value of the object is passed	Not allowed under LE/VSE
Indirect	A pointer points to the value of an object	A pointer points to the address of an object

Figure 19 illustrates these argument-passing styles. In Figure 19, register 1 (R1) points to the value of an object, a pointer to the value of an object, or a pointer to the address of an object.

By Value (Direct)

By Value (Indirect)



By Reference (Indirect)

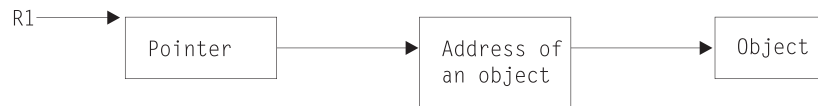


Figure 19. Argument-Passing Styles in LE/VSE

HLL semantics usually determine when data is passed by value or by reference. LE/VSE supports argument-passing styles as shown in Table 15.

Table 15. Default Passing Style per HLL

Language	Default Argument
C	By Value (Direct)
COBOL	By Reference (Indirect) (COBOL BY REFERENCE) By Value (Indirect) (COBOL BY CONTENT)
PL/I	By Reference (Indirect)[1]

Notes:

1. However, when SYSTEM(CICS) or SYSTEM(DLI) is specified, PL/I VSE main procedures assume by value (direct) for parameters. (See “PL/I Argument-Passing Considerations” on page 55 for a discussion of OPTIONS(BYVALUE).)

   PL/I also supports by value (indirect) (also known as *by content*), which you can obtain by passing an argument in parentheses, for example, `A` in `CALL X((A), B)`.

Preparing Your Main Routine to Receive Parameters

When coding a main routine to receive a parameter list from the operating system, consider the following:

- The HLL in which your main routine is written.

  HLL semantics determine how you code your main routine in order to receive a parameter list.
- The method of main routine invocation.

  You should consider the environment (batch without DL/I, batch with DL/I, or CICS) in which your main routine is invoked.
- The compiler or run-time options that you must specify.

The settings of the C PLIST run-time option or the PL/I SYSTEM compiler option that you must specify are based on the environment (batch without DL/I, batch with DL/I, or CICS) in which your main routine is invoked, and you must specify different settings accordingly.

The following tables summarize the various options to consider when preparing a main routine to receive parameters in each environment:

For this type of application
Refer to

Batch without DL/I
Table 16

Batch with DL/I
Table 17 on page 53

CICS
Table 18 on page 54

Assembler calling HLL
Table 19 on page 54

The tables also provide sample coding for each HLL.

Table 16. Coding a Main Routine to Receive an Inbound Parameter List in Batch without DL/I

Language	Recommended Options Setting	Sample Main Routine Code

C	Specify PLIST(HOST) run-time option. If not specified, PLIST(HOST) is the default.	`main(int argc, char * argv[])`
		`{`
		`   ⋮`
		`     }`

COBOL	No specific options required.	`IDENTIFICATION DIVISION.`
		`   ⋮`
		`DATA DIVISION.`
		`   ⋮`
		`LINKAGE SECTION.`
		` 01 PARMDATA.`
		`  02 STRINGLEN   PIC 99 USAGE IS BINARY.`
		`  02 STR.`
		`   03 PARM-BYTE  PIC X OCCURS 0 TO 80`
		`                 DEPENDING ON STRINGLEN.`
		`   ⋮`
		`PROCEDURE DIVISION USING PARMDATA.`
		`   ⋮`

PL/I	Specify SYSTEM(VSE) compile-time option.	`*PROCESS SYSTEM(VSE);`
		` MYMAIN: PROC (A) OPTIONS (MAIN);`
		
		`         DCL  A CHAR(80) VARYING;`
		`   ⋮`

Table 17. Coding a Main Routine to Receive an Inbound Parameter List in Batch with DL/I

Language	Recommended Options Setting	Sample Main Routine Code

C	Specify PLIST(OS) and ENV(DLI) run-time option.	`#pragma runopts(env(dli),plist(os))`
		`#include <ims.h>`
		`typedef struct {PCB_STRUCT(10)} PCB_10_TYPE;`
		`main()`
		`{`
		`   PCB_STRUCT_8_TYPE *alt_pcb;`
		`   PCB_10_TYPE *db_pcb;`
		`   IO_PCB_TYPE *io_pcb;`
		`   ⋮`
		`}`

COBOL	No specific options required.	`IDENTIFICATION DIVISION.`
		`   ⋮`
		`DATA DIVISION.`
		`   ⋮`
		`   LINKAGE SECTION.`
		`   01 PCB1.`
		`      02 ... .`
		`   01 PCB2.`
		`      02 ... .`
		`   ⋮`
		`PROCEDURE DIVISION.`
		`ENTRY 'DLITCBL' USING PCB1, PCB2.`
		`   ⋮`

PL/I	Specify SYSTEM(DLI) compile-time option.	`*PROCESS SYSTEM(DLI);`
		` MYMAIN: PROC (X,Y,Z) OPTIONS(MAIN);`
		
		`         DCL (X,Y,Z) POINTER;`
		`         DCL 1 PCB based (X),`
		`   ⋮`

Table 18. Coding a Main Routine to Receive an Inbound Parameter List in CICS

Language	Recommended Options Setting	Sample Main Routine Code

C	Do not specify any PLIST option. argc = 1 and argv[0] = transaction id.	main(int argc,char *argv[])
		{
		⋮
		}

COBOL	No specific options required.	IDENTIFICATION DIVISION. ⋮ DATA DIVISION. ⋮ LINKAGE SECTION. 01 DFHEIBLK. ⋮ 01 DFHCOMMAREA. ⋮ PROCEDURE DIVISION USING DFHEIBLK DFHCOMMAREA. ⋮
PL/I	Specify SYSTEM(CICS) compile-time option.	*PROCESS SYSTEM(CICS); MYMAIN: PROC (DFHEIPTR, DFHCOMMAREAPTR_PTR) OPTIONS(MAIN); /*pointer to EIB*/ /*supplied by CICS translator*/ DCL DFHEIPTR POINTER; /*pointer to commarea*/ DCL DFHCOMMAREAPTR_PTR POINTER; ⋮

Table 19. Coding a Main Routine to Receive an Inbound Parameter List in Batch. Method of invocation: Assembler passing an arbitrary parameter list that LE/VSE is not to interpret.

Language	Recommended Options Setting	Sample Main Routine Code
C	Specify PLIST(OS) run-time option.	main() { access register 1 through __osplist; ⋮ }
COBOL	No specific options required.	IDENTIFICATION DIVISION. ⋮ DATA DIVISION. ⋮ LINKAGE SECTION. 01 PARM1... 01 PARM2... ⋮ PROCEDURE DIVISION USING PARM1, PARM2. ⋮
PL/I	Specify SYSTEM(VSE) compile-time option and NOEXECOPS procedure option.	*PROCESS SYSTEM(VSE); MYMAIN: PROC (PARM1,PARM2,...) OPTIONS (MAIN NOEXECOPS); DCL PARM1... DCL PARM2... ⋮

PL/I Argument-Passing Considerations

The PL/I OPTIONS option of both the PROCEDURE statement and ENTRY declaration permits you to specify the mutually exclusive options BYVALUE and BYADDR.

OPTIONS(BYVALUE)
: Specifies that the PL/I procedure expects arguments to be passed to it by value (direct). OPTIONS(BYVALUE) can be specified for external PROCEDURE statements and ENTRY declarations. It applies to all arguments and argument descriptors.

OPTIONS(BYADDR)
: Specifies that the PL/I procedure expects arguments to be passed to it by reference (indirect) or by value (indirect). OPTIONS(BYADDR) can be specified for external PROCEDURE statements and for ENTRY declarations. It applies to all arguments and argument descriptors.

OPTIONS(BYVALUE) cannot be specified for the following constructs:

- ENTRY statements:

```
ENTRY(N) OPTIONS(BYVALUE);                         /* invalid */
```

- Declaration of a parameter:

```
PROC(ARG1);
DCL ARG1 FIXED BIN(31) BYVALUE;                    /* invalid */
```

- Parameter descriptor in an ENTRY declaration:

```
DCL T ENTRY(FIXED BIN(31) BYVALUE) EXTERNAL;          /* invalid */
```

All parameters, parameter descriptors, or return values must be specified with either the POINTER or FIXED BIN(31) data type. Return values are passed back in register 15.

OPTIONS(BYADDR) is the default unless the external procedure specifies OPTIONS(MAIN) and is compiled with the SYSTEM(CICS) or SYSTEM(DLI) compiler option. In this case, OPTIONS(BYVALUE) is the default. In general, you should specify OPTIONS(BYVALUE) only for a main procedure compiled with the SYSTEM(CICS) or SYSTEM(DLI) compiler option.

OPTIONS(BYVALUE) for a main procedure implies OPTIONS(NOEXECOPS).

PL/I does not support calls to routines that modify the body of an indirect argument list built by PL/I compiled code.

Chapter 7. Routines That Must Be Reentrant

This chapter shows you how to make your application reentrant. *Reentrancy* allows more than one user to share a single copy of an executable phase. If your application is not reentrant, each application that calls your application must load a separate copy of your application.

Understanding the Basics

The following applications **must** be reentrant:
- Routines to be loaded into the shared virtual area (SVA)
- Routines to be used with CICS

Your routine should be reentrant if it is a large routine that is likely to have multiple concurrent users. Less storage is used if multiple users share the routine concurrently. Reentrancy also offers some performance enhancement because there is less paging to auxiliary storage.

If you want your routine to be reentrant, ensure that it does not alter any static storage that is part of the executable phase. If the static storage is altered, the routine is not reentrant and its results are unpredictable.

Making Your C Program Reentrant

Under C, reentrant programs can be categorized by their reentrancy type as follows:

Natural reentrancy
: The attribute of programs that contain no static external data (also known as *writable storage*), that is, data that is external to the program.

Constructed reentrancy
: The attribute of applications that contain external data and require additional processing to become reentrant.

Natural Reentrancy

A C program is naturally reentrant if it contains no external data. In C, the following are considered external data:
- Variables using the `extern` storage class
- Variables using the `static` storage class
- Writable strings

If your program contains no external data, compile it as you would normally and install it in one of the locations listed in “Installing a Reentrant Phase” on page 58.

Constructed Reentrancy

Constructed reentrancy is achieved by running the object module produced by the C compiler through the prelinker. The prelinker (described in more detail in Chapter 3, “Prelinking an Application,” on page 11) concatenates compile-time initialization information from one or more object modules into a single initialization unit.

© Copyright IBM Corp. 1991, 2005

Programs with constructed reentrancy are split into two parts:
- A variable or nonreentrant part that contains external data
- A constant or reentrant part that contains executable code and constant data

Each user running the program receives a private copy of the first part, which is mapped by the prelinker and is initialized at run time. The second part can be shared across multiple partitions only if it is installed in the SVA.

Generating a Reentrant C Object Module

To generate a reentrant C object module, follow these steps:

1. If your program contains external data, compile your C source files using the RENT compile-time option. See *LE/VSE C Run-Time Programming Guide* for more information on RENT.
2. Use the prelinker to combine all input object modules into a single object module.

   Note: You cannot run an object module through the prelinker more than once. You also can only link on the same platform you prelinked on.
3. To get the greatest benefit from reentrancy, link edit the object module and install the phase in the SVA.

Making Your COBOL Routine Reentrant

If you intend to have multiple users execute a COBOL routine at the same time, make it reentrant by specifying the RENT option when you compile it. For information about specifying the RENT compiler option, see *IBM COBOL for VSE/ESA Programming Guide.*

Making Your PL/I Routine Reentrant

If you intend to have multiple users execute a PL/I routine at the same time, make it reentrant by specifying the REENTRANT procedure option when you code it. For information about specifying the REENTRANT procedure option, see *IBM PL/I for VSE/ESA Language Reference.*

Installing a Reentrant Phase

You will get the most benefits from reentrancy if you install your link-edited phase in the SVA.

Your routine runs correctly if it is not installed in the SVA, but you can save storage by installing the phase in the SVA.

Note: Installing a phase in the SVA requires that the system directory list (SDL) be updated.

Part 3. Concepts, Services, and Models

Chapter 8. Initialization and Termination Under LE/VSE . . . 63
Understanding the Basics . . . 63
LE/VSE Initialization . . . 65
What Happens During Initialization . . . 65
LE/VSE Termination . . . 66
What Causes Termination . . . 66
What Happens During Termination . . . 67
Thread Termination . . . 67
Enclave Termination . . . 67
Process Termination . . . 68
Managing Return Codes in LE/VSE . . . 68
How the LE/VSE Enclave Return Code is Calculated . . . 68
PL/I Considerations . . . 69
Setting and Altering User Return Codes . . . 69
For C . . . 69
For COBOL . . . 69
For PL/I . . . 69
How the Enclave Reason Code is Calculated 70
Termination Behavior for Unhandled Conditions . . . 70
Determining the Abend Code . . . 71
Abend Codes Generated by CEEBXITA . . . 71
Abnormal Termination Messages and Abend Codes Generated by ABTERMENC(ABEND) Run-Time Option . . . 72
Program Interrupt Codes . . . 73

Chapter 9. Program Management Model . . . 75
Understanding the Basics . . . 75
Program Management Model Terminology . . . 75
LE/VSE Terms and Their HLL Equivalents . . . 75
Terminology for Data . . . 76
Processes . . . 77
Enclaves . . . 77
The Enclave Defines the Scope of Language Semantics . . . 77
Additional Enclave Characteristics . . . 78
Threads . . . 79
The Full Language Environment Program Management Model . . . 79

Chapter 10. Stack and Heap Storage . . . 81
Understanding the Basics . . . 81
Stack Storage Overview . . . 83
Tuning Stack Storage . . . 84
COBOL Considerations . . . 84
PL/I Storage Considerations . . . 84
Heap Storage Overview . . . 85
Heap IDs Recognized by the LE/VSE Heap Manager . . . 86
AMODE Considerations for Heap Storage . . . 87
Tuning Heap Storage . . . 87
COBOL Considerations . . . 87
Storage Performance Considerations . . . 87
COBOL and LE/VSE Storage Considerations . . . 87
Dynamic Storage Services . . . 89
Examples of Callable Storage Services . . . 90
C Example of Building a Linked List . . . 90
COBOL Example of Building a Linked List . . . 92
PL/I Example of Building a Linked List . . . 94
C Example of Storage Management . . . 96
COBOL Example of Storage Management . . . 98
PL/I Example of Storage Management . . . 100

Chapter 11. LE/VSE Condition Handling Introduction . . . 103
Understanding the Basics . . . 103
Related Run-Time Options and Callable Services 104
The Stack Frame Model . . . 105
The Handle Cursor . . . 106
The Resume Cursor . . . 106
What Is a Condition in LE/VSE? . . . 106
Steps in Condition Handling . . . 107
Enablement Step . . . 107
TRAP Effects on the Condition Handling Process . . . 108
LE/VSE Abends and the Enablement Step 108
Using XUFLOW and CEE5SPM to Enable and Disable Hardware Conditions . . . 108
Condition Step . . . 109
Influencing Condition Handling with the ERRCOUNT Run-Time Option . . . 110
Termination Imminent Step . . . 111
Processing the T_I_U Condition . . . 112
Processing the T_I_S Condition . . . 112
The Termination Imminent Step and the TERMTHDACT Run-Time Option . . . 113
CEESGL and the Termination Imminent Step 114
Invoking Condition Handlers . . . 114
Responses to Conditions . . . 116
Condition Handling Scenarios . . . 116
Scenario 1: Simple Condition Handling . . . 116
Scenario 2: Condition Handling with User-Written Condition Handler Present for T_I_U . . . 118
Scenario 3: Condition Handling with User-Written Condition Handler Present for Divide-by-Zero Condition . . . 119

Chapter 12. LE/VSE and HLL Condition Handling Interactions . . . 121
Understanding the Basics . . . 121
C Condition Handling Semantics . . . 121
Comparison of C-LE/VSE Terminology . . . 122
Controlling Condition Handling in C . . . 122
Using the signal() Function . . . 123
Using the raise() Function . . . 123
C atexit() Considerations . . . 123
C Condition Handling Actions . . . 124
C Condition Handling Examples . . . 125
C Signal Representation of S/370 Exceptions 127

© Copyright IBM Corp. 1991, 2005

COBOL Condition Handling Semantics 128
COBOL Condition Handling Examples 129
Restrictions about Resuming Execution after an IGZ Condition Occurs 131
IGZ Condition of Severity 2 or Greater . . . 131
COBOL STOP RUN Statement 131
Reentering COBOL Programs after Stack Frame Collapse 131
Handling Fixed-Point and Decimal Overflow Conditions 132
PL/I Condition Handling Semantics 132
PL/I Condition Handling Actions 132
Promoting Conditions to the PL/I ERROR Condition 133
Mapping Non-PL/I Conditions to PL/I Conditions 134
Additional PL/I Condition Handling Considerations 134
PL/I Condition Handling Example 135

Chapter 13. Coding a User-Written Condition Handler 137
Understanding the Basics 137
Types of Conditions You Can Handle 137
User-Written Condition Handler Interface using CEEHDLR 138
Registering a User-Written Condition Handler using USRHDLR 139
Nested Conditions. 140
Nested Conditions in Applications Containing a COBOL Program 140
Using LE/VSE Condition Handling with Nested COBOL Programs 141
Examples with a Registered User-Written Condition Handler 141
Handling a Divide-by-Zero Condition in C or COBOL 141
C Handling a Divide-by-Zero Condition . . 143
COBOL Handling a Divide-by-Zero Condition 145
Handling an Out-of-Storage Condition in C or COBOL 148
C Examples Using CEEHDLR, CEEGTST, CEECZST, and CEEMRCR 149
COBOL Examples Using CEEHDLR, CEEGTST, CEECZST, and CEEMRCR . . . 153
Signaling and Handling a Condition in a C Routine 158
Handling a Divide-by-Zero Condition in a COBOL Program 160
Handling a Program Check in an Assembler Routine 165

Chapter 14. Using Condition Tokens 171
Understanding the Basics 171
Understanding the Structure of the Condition Token 172
The Effect of Coding the fc Parameter 173
Testing a Condition Token for Success 174
Testing Condition Tokens for Equivalence . . . 174
Testing Condition Tokens for Equality 175
The Effect of Omitting the fc Parameter 175
Using Symbolic Feedback Codes 175
Locating Symbolic Feedback Codes for Conditions 175
Including Symbolic Feedback Code Files . . . 176
Examples Using Symbolic Feedback Codes . . 178
C 178
COBOL 179
PL/I 181
Condition Tokens for C Signals under C 182
LE/VSE-provided q_data Structure for Abends . . 182

Chapter 15. Using and Handling Messages . . 185
Understanding the Basics 185
Creating Messages. 185
Creating a Message Source File 186
Using the CEEBLDTX Utility 189
Files Created by CEEBLDTX 190
Running the CEEBLDTX Utility 191
Running the CEEBLDTX Utility on VSE . . 191
Running the CEEBLDTX Utility on CMS . . 192
Assembling and Link-Editing the Message File 192
CEEBLDTX Error Messages 192
Creating a Message Module Table 195
Assigning Values to Message Inserts. 196
Using Messages in Code. 197
Interpreting Run-Time Messages 198
Specifying National Language 199
Handling Message Output 199
Using LE/VSE MSGFILE 199
Using C Input/Output Functions. 200
Using COBOL Input/Output Statements . . . 201
Using PL/I Input/Output Statements 203
MSGFILE Considerations When Using PL/I . . 204
Examples Using Multiple Message Handling Callable Services 205
C Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG 205
COBOL Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and CEEMSG 207
PL/I Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG 210

Chapter 16. Using Date and Time Services . . 213
Understanding the Basics 213
Working with Date and Time Services 214
Date Limits 214
Picture Character Terms and Picture Strings . . 215
Notation for Eras 215
Performing Calculations on Date and Time Values 216
Century Window Routines 216
National Language Support for Date and Time Services 217
Examples Using Date and Time Callable Services 217
Examples Illustrating Calls to CEEQCEN and CEESCEN 219
Calls to CEEQCEN and CEESCEN in C . . 219
Calls to CEEQCEN and CEESCEN in COBOL 220

Calls to CEEQCEN and CEESCEN in PL/I 221
Examples Illustrating Calls to CEESECS . . . 222
Calls to CEESECS in C 222
Calls to CEESECS in COBOL 223
Calls to CEESECS in PL/I 225
Examples Illustrating Calls to CEESECS and CEEDATM 226
Calls to CEESECS and CEEDATM in C . . . 226
Calls to CEESECS and CEEDATM in COBOL 228
Calls to CEESECS and CEEDATM in PL/I 230
Examples Illustrating Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM. 231
Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in C 231
Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in COBOL 233
Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in PL/I 236
Example Illustrating Calls to CEEDAYS, CEEDATE, and CEEDYWK. 238
Calls to CEEDAYS, CEEDATE, and CEEDYWK in C 238
Calls to CEEDAYS, CEEDATE, and CEEDYWK in COBOL 240
Calls to CEEDAYS, CEEDATE, and CEEDYWK in PL/I 243
Calls to CEECBLDY in COBOL 245

Chapter 17. National Language Support Services 247
Understanding the Basics 247
Setting the National Language. 248
Setting the Country Code 248
Euro Support 249
Combining National Language Support and Date and Time Services 249
Calls to CEE5CTY, CEEFMDT, and CEEDATM in C 249
Calls to CEE5CTY, CEEFMDT, and CEEDATM in COBOL 252
Example Using CEE5CTY, CEEFMDT, and CEEDATM in PL/I 254

Chapter 18. Locale Callable Services 257
Understanding the Basics 257
Developing Internationalized Applications. . . . 258
Examples of Using Locale Callable Services . . . 258
Examples Illustrating Calls to CEEFMON . . . 258
Calls to CEEFMON in COBOL 259
Calls to CEEFMON in PL/I 260
Examples Illustrating Calls to CEEFTDS . . . 261
Calls to CEEFTDS in COBOL 261
Calls to CEEFTDS in PL/I 263
Examples Illustrating Calls to CEELCNV and CEESETL. 264
Calls to CEELCNV and CEESETL in COBOL 264
Calls to CEELCNV and CEESETL in PL/I 266
Examples Illustrating Calls to CEEQDTC and CEESETL. 267
Calls to CEEQDTC and CEESETL in COBOL 267
Calls to CEEQTDC and CEESETL in PL/I 269
Examples Illustrating Calls to CEESCOL . . . 270
Calls to CEESCOL in COBOL 270
Calls to CEESCOL in PL/I 272
Examples Illustrating Calls to CEESETL and CEEQRYL 273
Calls to CEESETL and CEEQRYL in COBOL 273
Calls to CEESETL and CEEQRYL in PL/I . . 275
Examples Illustrating Calls to CEEQRYL and CEESTXF. 276
Calls to CEEQRYL and CEESTXF in COBOL 276
Calls to CEEQRYL and CEESTXF in PL/I 278

Chapter 19. General Callable Services 281
CEE5DMP Callable Service 281
CEE5PRM Callable Service 282
CEE5PRML Callable Service 282
CEE5TSTG Callable Service. 282
CEE5USR Callable Service 282
CEEGPID Callable Service 282
CEERAN0 Callable Service 283
CEETEST Callable Service 283
Examples of Using Basic Callable Services. . . . 283

Chapter 20. Math Services 287
Understanding the Basics 287
Call Interface to Math Services 289
Parameter Types: parm1 Type and parm2 Type 289
Examples of Calling Math Services 290
Calling CEESSLOG in C. 290
Calling CEESSLOG in COBOL. 291
Calling CEESSLOG in PL/I. 292

Chapter 8. Initialization and Termination Under LE/VSE

This chapter describes initialization and termination under LE/VSE. It describes how you can customize your applications during initialization and termination by using LE/VSE run-time options, callable services, and user exits. It includes instructions on how to use return codes, abend codes, and VSE cancel codes to respond to initialization and termination actions, as well as to conditions that remain unhandled.

Understanding the Basics

Initialization and termination establish the state of various parts of the LE/VSE program management model that supports multi-language applications. The program management model describes three major entities of a program structure:

Process
A collection of resources (code and data).

Enclave
A collection of program units consisting of at least one main routine.

Thread
The basic unit of execution.

When you run a routine, LE/VSE initializes the run-time environment by creating a process, an enclave, and an initial thread. You can modify initialization by running a user exit, written either in assembler or in an HLL.

During termination, threads, enclaves, and processes are terminated. Through LE/VSE's run-time options and callable services for termination, you can control how a thread, enclave, or process terminates. For example, you can control whether an abend or a return code is generated from an application that terminates with an unhandled condition of severity 2 or greater.

© Copyright IBM Corp. 1991, 2005

Related Options and Services

Run-Time Options

ABTERMENC
: Specifies whether an enclave terminates with an abnormal termination message (non-CICS environment) or abend (CICS environment), or with a return code and a reason code when there is an unhandled condition of severity 2 or greater

TERMTHDACT
: Specifies the level of information that you want to receive after an unhandled condition of severity 2 or greater causes a thread to terminate

Callable Services

CEE5ABD
: Terminates an enclave with or without clean-up

CEE5GRC
: Returns the user enclave return code to your routine (along with CEE5SRC, it allows you to use return code-based programming techniques)

CEE5PRM and CEE5PRML
: Returns to your routine the parameter string specified when your application was invoked

CEE5SRC
: Sets the user enclave return code, which is used to calculate the final enclave return code at termination

User Exits

CEEBXITA
: An assembler user exit for enclave initialization, and enclave and process termination

CEEBINT
: An HLL user exit (written in C, PL/I, or LE/VSE-conforming assembler) called at enclave initialization

See Chapter 25, “Using Run-Time User Exits,” on page 319 for more information on user exits.

Preinitialization Interface

CEEPIPI
: CEEPIPI performs various initialization functions

See Chapter 27, “Using Preinitialization Services,” on page 363 for more information on the preinitialization interface.

See *LE/VSE Programming Reference* for syntax information on run-time options and callable services.

LE/VSE Initialization

During initialization, a process, an enclave, and then an initial thread are created. You can affect initialization at the enclave level, by using either the assembler or HLL user exits.

Process initialization sets up the framework to manage enclaves and initializes resources that can be shared among enclaves. Enclave initialization creates the framework to manage enclave-related resources and the threads that run within the enclave. Thread initialization acquires a stack and enables the condition manager for the thread.

What Happens During Initialization

When you run an application under LE/VSE, the following sequence of events occurs:

1. LE/VSE runs the assembler user exit CEEBXITA.

   CEEBXITA runs prior to initialization of the enclave. You can modify the environment in which your application runs by:
   - Specifying run-time options
   - Listing VSE cancel codes, program-interruption codes, and user abend codes to be exempted from normal LE/VSE condition handling
   - Checking the values of program arguments

   IBM provides a default version of CEEBXITA and several samples you can use to customize your application to perform tasks such as enforcing a set of run-time options for a particular environment. Because CEEBXITA runs before any HLLs have been established, it is written in assembler language so that it can establish parameters such as stack size and trap settings for the HLLs.

   CEEBXITA can function as an application-specific or installation-wide exit. If you customize CEEBXITA to do application-specific processing (for example, checking values of program arguments), you must link the exit with the application phase. (Conversely, installation-wide user exits must be linked with the LE/VSE initialization library routines.)

   An application-specific user exit has priority over an installation-wide exit, so you can customize a user exit for a particular application without affecting the installation default version.

   For more information on the function and location of the CEEBXITA user exit, see Chapter 25, “Using Run-Time User Exits,” on page 319.
2. LE/VSE examines the phase and initializes all languages identified in the application.

   Under LE/VSE, an interlanguage communication (ILC) application works as shown in Figure 20 on page 66. Because all the language conventions are already established and do not need to be initialized and terminated between calls to other routines, the processing is significantly faster when using LE/VSE-conforming HLLs.

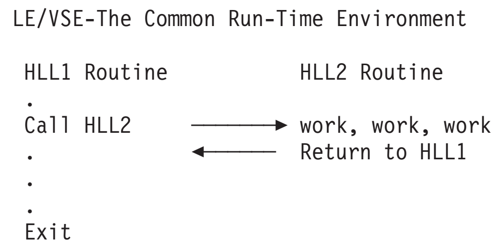


Figure 20. LE/VSE ILC—Only One Run-Time Environment to Initialize

Performance Consideration: LE/VSE initializes all languages included in an application, regardless of whether all of them are used. To optimize performance, include only those languages your application actually uses.

3. LE/VSE runs the HLL user exit CEEBINT.

   CEEBINT lets you perform tasks such as recording accounting statistics or calling other user exits. You can write a customized version of CEEBINT in any LE/VSE-conforming language except COBOL. COBOL applications can, however, use CEEBINT written in another language.

   IBM provides an object module default version of CEEBINT that consists simply of an immediate return to the application. This default version is automatically link-edited with your application unless you provide a customized version of CEEBINT.

   For more information on the function and location of the CEEBINT user exit, see Chapter 25, "Using Run-Time User Exits," on page 319.

LE/VSE Termination

LE/VSE termination provides services that restore the operating environment to its original state after your application either runs to completion or terminates abnormally. You can affect termination through the use of run-time options, callable services, and user exits. You can also decide whether to use a return code at termination or an abend, if a condition is unhandled with a severity code of 2 or greater (see "Termination Behavior for Unhandled Conditions" on page 70 for more information).

What Causes Termination

Under LE/VSE, an application terminates when any of the following conditions occur:

- The last thread in the enclave terminates (which in turn terminates the enclave).
- An HLL construct issues a request for the termination of an enclave, for example:
  - C's `abort()` function
  - C's `raise(SIGTERM)` function
  - C's `exit()` function
  - COBOL's STOP RUN statement
  - PL/I's STOP or EXIT statement
- The main routine in the enclave returns to its caller; that is, an implicit STOP is performed. For example:
  - COBOL's GOBACK statement in a main program
  - PL/I's END or RETURN statement in a main procedure
  - The end of a C `main()` function

- An abend is requested by the application (that is, the application calls CEE5ABD).
- An unhandled condition of severity 2 or greater occurs (see "Termination Behavior for Unhandled Conditions" on page 70).

What Happens During Termination

The following sequence of events occurs during termination:

1. C `atexit()` functions are invoked, if present. They are not invoked if abnormal termination occurs.
2. PL/I FINISH ON-units are invoked if established.
3. For normal termination, the enclave return code is set (see "Managing Return Codes in LE/VSE" on page 68). For abnormal termination caused by an unhandled condition of severity 2 or greater, either a return code and reason code is returned, or an abnormal termination message (non-CICS environment) or abend (CICS environment) is generated, based on settings specified in CEEBXITA (see "Termination Behavior for Unhandled Conditions" on page 70).
4. CEEBXITA is invoked for enclave termination after all application code has completed, but before any enclave resources are relinquished.

   You can modify CEEBXITA to request an abend and a dump. Because the environment is still active, the dump accurately reflects the state of the environment before an enclave is terminated.
5. The environment is terminated:
   - All enclaves are terminated
   - All enclave resources are returned to the operating system
   - Any files that LE/VSE manages are closed
   - The debug tool is terminated, if active
6. CEEBXITA is invoked for process termination after the environment is terminated. You can modify CEEBXITA to close files, request an abend, or request a dump. A dump requested at this point, however, does not have the degree of detail that one requested during enclave termination has.

   CEEBXITA is not invoked for process termination if there is an unhandled condition of severity 2 or greater, or if CEEBXITA requests an abend during enclave termination.

   For more information on the CEEBXITA assembler user exit, see Chapter 25, "Using Run-Time User Exits," on page 319.

Depending on the setting of the TERMTHDACT run-time option, you might receive a message, a trace of the active routines, or a dump when a condition of severity 2 or greater occurs. See *LE/VSE Programming Reference* for more information on TERMTHDACT.

Thread Termination

A thread terminating is analogous to an enclave terminating, because LE/VSE Version 1 Release 4 supports only single threads. See "Enclave Termination" for information on enclave termination.

Enclave Termination

When an enclave terminates, LE/VSE releases resources allocated on behalf of the enclave and performs various other activities including the following:

- Calls HLL-specific termination routines for HLLs that were active during the running of the program
- Runs LE/VSE user-written condition handlers, if present

- Deletes phases loaded by LE/VSE
- Frees all storage obtained by LE/VSE services
- Calls the CEEBXITA assembler user exit for enclave termination
- Frees LE/VSE control blocks for the enclave
- Depending on the setting in the HLL or assembler user exit, LE/VSE sets a return code and reason code or generates an abnormal termination message (non-CICS environment) or abend (CICS environment)
- Restores the program mask and registers to preinitialization values
- Returns control to the enclave creator

Process Termination

Process termination occurs when the last enclave in the process terminates. Process termination deletes the structure that kept track of the enclaves within the process, releases the process control block (PCB) and associated resources, and returns control to the creator of the process.

Because LE/VSE Version 1 Release 4 generally supports a single enclave running within a single process, termination of the enclave means that your application has terminated. For exceptions to the single enclave within a single process and an enclave return and reason code being returned to the invoker, see Chapter 28, "Using Nested Enclaves," on page 393.

LE/VSE explicitly relinquishes all resources it obtains. Routines that obtain resources directly from the host system must explicitly relinquish the resource. If these resources are not explicitly released, the environment can be corrupted because LE/VSE has no method for releasing these resources.

Managing Return Codes in LE/VSE

This section discusses how LE/VSE calculates and uses return codes and reason codes during enclave termination. (The return codes between subroutine calls that are implemented with programming language constructs are addressed in the appropriate language-specific programming guides.)

Before LE/VSE, some HLLs handled conditions that occur in the run-time environment by using a *return code*-based model. Such a model typically allows return codes to be passed between called subroutines and from the main routine back to the operating system to communicate the status of requested operations. LE/VSE, on the other hand, uses a *condition*-based model to communicate conditions, as described in Chapter 14, "Using Condition Tokens," on page 171.

Although LE/VSE supports applications that rely on passing return codes from called subroutines and checking these return codes, you are encouraged to use LE/VSE condition handling mechanisms, such as user-written condition handlers, instead.

How the LE/VSE Enclave Return Code is Calculated

When an enclave terminates, LE/VSE provides an LE/VSE enclave return code and an enclave reason code (sometimes called a return code modifier). The LE/VSE enclave return code is calculated by summing the user return code generated by the HLL (see "Setting and Altering User Return Codes" on page 69) and the enclave reason code (see "How the Enclave Reason Code is Calculated" on page 70) as follows:

```
LE/VSE enclave return code = user return code + enclave reason code
```

The LE/VSE enclave return code is placed in register 15, and the enclave reason code is placed in register 0.

PL/I Considerations

The severities of some PL/I conditions have been redefined from what they were in the pre-LE/VSE-conforming version of PL/I. See *IBM PL/I for VSE/ESA Migration Guide* for detailed information on the changes.

Setting and Altering User Return Codes

User return codes can be set and altered by the CEE5SRC callable service and by language constructs. The user return code value is based on the reason an enclave terminates and the language of the routine that initiates termination, as follows:

For C

When a C routine terminates an enclave normally by reaching the end of the `main()` function, the user return code value is 0 (assuming the return type of the `main()` function is `int`). When a C routine terminates an enclave with a language construct such as `exit(`*n*`)` or `return(`*n*`)`, the value of *n* is used. In either case, any user return codes set through CEE5SRC are ignored; likewise, in an ILC application, any user return codes set with PL/I language constructs are also ignored.

If the enclave terminates due to an unhandled condition of severity 2 or greater, the user return code value used is the last one set by either CEE5SRC or, in an ILC application, PL/I language constructs. If neither CEE5SRC nor PL/I language constructs set the user return code, the user return code value is 0. See “Termination Behavior for Unhandled Conditions” on page 70 for information on unhandled conditions.

See *LE/VSE C Run-Time Programming Guide* for more information about C language constructs.

For COBOL

When a COBOL program initiates enclave termination, such as with a STOP RUN statement in a subprogram or a GOBACK statement in a main program, the user return code value is taken from the RETURN-CODE special register; any user return codes set through CEE5SRC are ignored. Likewise, in an ILC application, any user return codes set with PL/I language constructs are also ignored. Thus, you can set and alter the user return code and pass it across program boundaries in register 15. See *IBM COBOL for VSE/ESA Programming Guide* for more details on the RETURN-CODE special register and COBOL language constructs.

If the enclave terminates due to an unhandled condition with severity 2 or greater, the RETURN-CODE special register is not used in the enclave return code calculation. Instead, the user return code value used is the last one set by either CEE5SRC or, in an ILC application, PL/I language constructs. If neither CEE5SRC nor PL/I language constructs have been used to set the user return code, the user return code value is 0. See “Termination Behavior for Unhandled Conditions” on page 70 for information on unhandled conditions.

For PL/I

You can set and alter the user return code with the PLIRETC function or the OPTIONS(RETCODE) attribute. The PLIRETV function retrieves the current value of the user return code.

When a PL/I routine initiates enclave termination, such as with a STOP or EXIT statement in a subroutine or with a RETURN or END statement in a main procedure, the user return code is the value set with the PLIRETC function or the OPTIONS(RETCODE) attribute. However, CEE5SRC can alter the user return code set with PLIRETC or the OPTIONS(RETCODE) attribute. If CEE5SRC was the last method used to set the user return code, the value set by CEE5SRC is used as the user return code.

If the enclave terminates due to an unhandled condition with severity 2 or greater, the user return code value set last (with either PL/I constructs or CEE5SRC) is used in the calculation of the enclave return code; if one has not been set, the user return code value is 0. See "Termination Behavior for Unhandled Conditions" for information on unhandled conditions.

See *IBM PL/I for VSE/ESA Language Reference* for details on PL/I language constructs.

How the Enclave Reason Code is Calculated

The enclave reason code provides additional information in support of the enclave return code. LE/VSE calculates the enclave reason code by multiplying a severity code that indicates how an enclave terminated by 1000.

The severity code is initially set to 0, indicating normal enclave termination. If the Termination_Imminent due to STOP (T_I_S) condition is signaled, it is set to 1. If the enclave terminates due to an unhandled condition of severity 2 or greater, the enclave reason code is set according to the severity of the unhandled condition that caused the enclave to terminate, as shown in Table 20. For more information about LE/VSE conditions and severity codes, see Table 28 on page 111.

Table 20 contains a summary of the enclave reason code produced when an enclave terminates. The condition severity column indicates the reason code for the original condition.

Table 20. Summary of Enclave Reason Codes

Condition Severity	Meaning	Enclave Reason Code
0	Normal application termination	0
Severity 1 condition	Termination_Imminent due to STOP	1000
Unhandled severity 2 condition	Error — abnormal termination	2000
Unhandled severity 3 condition	Severe error — abnormal termination	3000
Unhandled severity 4 condition	Critical error — abnormal termination	4000

Termination Behavior for Unhandled Conditions

When there is an unhandled condition of severity 2 or greater, you can choose whether an enclave terminates with an abnormal termination message (non-CICS environment) or abend (CICS environment), or with a return code and a reason code. LE/VSE will assign an abend code and return and reason codes, as described in this section, or you can assign values yourself, as described in "Setting and Altering User Return Codes" on page 69.

See Table 28 on page 111 for a discussion of conditions and how they are handled in LE/VSE.

Some users, especially those using COBOL, expect to receive an abend when an error is detected rather than a return code and a reason code. To get this behavior, they can use the ABTERMENC(ABEND) run-time option discussed in "Abnormal Termination Messages and Abend Codes Generated by ABTERMENC(ABEND) Run-Time Option" on page 72. Other users, however, expect to receive a return code and a reason code when there is an error.

If you are running in a CICS environment, the IBM-supplied default is to terminate the enclave with an abend for unhandled conditions of severity 2 or greater.

If you are running in a non-CICS environment, the IBM-supplied default is to terminate with a return code and reason code for unhandled conditions of severity 2 or greater. If you want the enclave to terminate with an abend, you can use the ABTERMENC(ABEND) run-time option or the CEEBXITA assembler user exit. The default version of CEEBXITA for non-CICS environments requests that the enclave terminate with a return code and a reason code.

Table 21 shows the various types of enclave termination that occur based on the ABTERMENC run-time option settings and the CEEAUE_ABND flag settings of CEEBXITA. See "CEEBXITA Assembler User Exit Interface" on page 323 for an explanation of the CEEAUE_ABND flag.

Table 21. Termination Behavior for Unhandled Conditions of Severity 2 or Greater

ABTERMENC Suboption	Value of CEEAUE_ABND Flag Enclave Termination	Enclave Termination Type
RETCODE	0	Return to caller with return code and reason code
RETCODE	1	Abend using CEEAUE_RETURN and CEEAUE_REASON
ABEND	0	Abend using the abnormal termination messages (non-CICS environment) or abend codes (CICS environment) listed in Table 23 on page 72
ABEND	1	Abend using CEEAUE_RETURN and CEEAUE_REASON

Determining the Abend Code

You can choose the abend code you want LE/VSE to use, based on whether the abend is requested by the assembler user exit or whether the ABTERMENC(ABEND) run-time option is used.

Abend Codes Generated by CEEBXITA

When you request an abend through CEEBXITA, the values contained in certain fields of the exit are used for the abend code and the reason code. Table 22 shows the abend codes used by LE/VSE when CEEBXITA requests an abend and does not modify the CEEAUE_RETURN code field.

Table 22. Abend Codes Used by LE/VSE when the Assembler User Exit Requests an Abend

Condition Severity	User Return Code	Abend Code in VSE	Abend Code in CICS
2	0	Abnormal termination message CEE3322C, user abend 2000	Transaction abend 2000

Table 22. Abend Codes Used by LE/VSE when the Assembler User Exit Requests an Abend (continued)

Condition Severity	User Return Code	Abend Code in VSE	Abend Code in CICS
3	0	Abnormal termination message CEE3322C, user abend 3000	Transaction abend 3000
4	0	Abnormal termination message CEE3322C, user abend 4000	Transaction abend 4000

Abnormal Termination Messages and Abend Codes Generated by ABTERMENC(ABEND) Run-Time Option

LE/VSE terminates the enclave with an abnormal termination message (non-CICS environment) or the same abend code that caused the unhandled condition of severity 2 or greater (CICS environment) if **both** of the following are true:

- You use the ABTERMENC(ABEND) run-time option.
- The assembler user exit does not alter the CEEAUE_ABND flag setting.

Table 23 shows the abnormal termination message, abend code, and reason code used when the enclave terminates due to the various unhandled conditions of severity 2 or greater and ABTERMENC(ABEND) is specified in both CICS and non-CICS environments.

Table 23. Abend Code Values Used by LE/VSE with ABTERMENC(ABEND)

Unhandled Condition	Abnormal Termination Message and/or Abend Code	Abend Reason Code[1]
System-generated abend	In an non-CICS environment, abnormal termination message CEE3321C and the VSE cancel code In a CICS environment, the original abend code	
User-generated abend	In an non-CICS environment, abnormal termination message CEE3322C and the original abend code In a CICS environment, the original abend code	In non-CICS environment, the original abend reason code
Program interrupt	In a non-CICS environment, abnormal termination message CEE3321C and the interruption code (see “Program Interrupt Codes” on page 73) In a CICS-environment, an ASRA abend code	
Software-raised condition	In a non-CICS environment, abnormal termination message CEE3322C and user abend code 4038 In a CICS environment, transaction 4038 abend	In a non-CICS environment, X'1'

Table 23. Abend Code Values Used by LE/VSE with ABTERMENC(ABEND) (continued)

Unhandled Condition	Abnormal Termination Message and/or Abend Code	Abend Reason Code[1]

Note:

1. In a CICS environment, when an abend is issued, only the abend code is returned. CICS does not return an abend reason code.

Program Interrupt Codes

A program interrupt can cause an unhandled condition of severity 2 or greater. When running with the ABTERMENC(ABEND) run-time option:

- In a CICS environment, an abend code of ASRA is issued for a program interrupt.
- In a non-CICS environment, abnormal termination message CEE3321C, containing VSE cancel code 20 and one of the program interrupt codes shown in Table 24, is issued for a program interrupt.

Table 24. Program Interrupt Codes in a Non-CICS Environment

Program Interrupt	Interrupt Code
Operation exception	01
Privileged operation exception	02
Execute exception	03
Protection exception	04
Segment translation exception (note 1)	04
Page translation exception (note 2)	04
Addressing exception	05
Specification exception	06
Data exception	07
Fixed-point overflow exception	08
Fixed-point divide exception	09
Decimal overflow exception	0A
Decimal divide exception	0B
Exponent overflow exception	0C
Exponent underflow exception	0D
Significance exception	0E
Floating-point divide exception	0F

Notes:

1. The operating system issues program interrupt code 04 for segment translation program interrupts.
2. The operating system issues program interrupt code 04 for page translation program interrupts.

Chapter 9. Program Management Model

Now that you have been introduced to how applications run in LE/VSE, you need to understand the Language Environment model, the model of program management under which LE/VSE operates. Understanding the model helps you recognize equivalent entities across LE/VSE-conforming programming languages and predict how your single- and mixed-language applications run. This chapter provides an overview of the Language Environment model.

The Language Environment program management model supports the language semantics of applications that run in the common run-time environment and defines the way routines or programs are put together to form an application.

LE/VSE Implementation Information

This release of LE/VSE implements a subset of the program management model. Features not supported in LE/VSE Version 1 Release 4 are clearly indicated in this manual.

Understanding the Basics

The Language Environment program management model has three basic entities—the process, enclave, and thread, each of which LE/VSE creates whenever you run a routine. This section describes each of these entities and their relationship to program management.

Program Management Model Terminology

Some terms used to describe the program management model are common programming terms; others have meanings that are specific to a given language. It is important that you understand the meaning of the program management model terminology LE/VSE uses and how it compares with existing languages. For more detailed definitions of these and other LE/VSE terms, please consult the "Language Environment Glossary" on page 417.

LE/VSE Terms and Their HLL Equivalents

Process
: The highest level of the Language Environment program management model; a collection of resources, both program code and data, consisting of at least one enclave.

Enclave
: The enclave defines the scope of HLL semantics. In LE/VSE, a collection of routines, one of which is designated as the main routine. The enclave contains at least one thread.

  Equivalent HLL terms: COBOL—run unit, C—program consisting of a main C function and its subroutines, PL/I—main procedure and all its subprocedures.

Thread
: An execution entity that consists of synchronous invocations and terminations of routines. The thread is the basic run-time path within the

© Copyright IBM Corp. 1991, 2005

Language Environment program management model; dispatched by the system with its own run-time stack, instruction counter, and registers.

Routine
In LE/VSE, either a procedure, function, or subroutine.

Equivalent HLL terms: COBOL—program, C—function, PL/I—procedure, BEGIN/END block.

Terminology for Data

Automatic data
Data that does not persist across calls; it is allocated with the same value on entry and reentry into a routine.

External data
Data with one or more named points by which the data can be referenced by other program units and data areas. External data is known throughout an enclave.

Local data
Data known only to the routine in which it is declared; equivalent to local data in C, WORKING-STORAGE in COBOL, and data with the PL/I INTERNAL attribute (whether implicitly, or by explicit declaration).

Figure 21 shows the simplest form of the Language Environment program management model and the resources that each component controls. Refer to the figure as you read about the program management model.

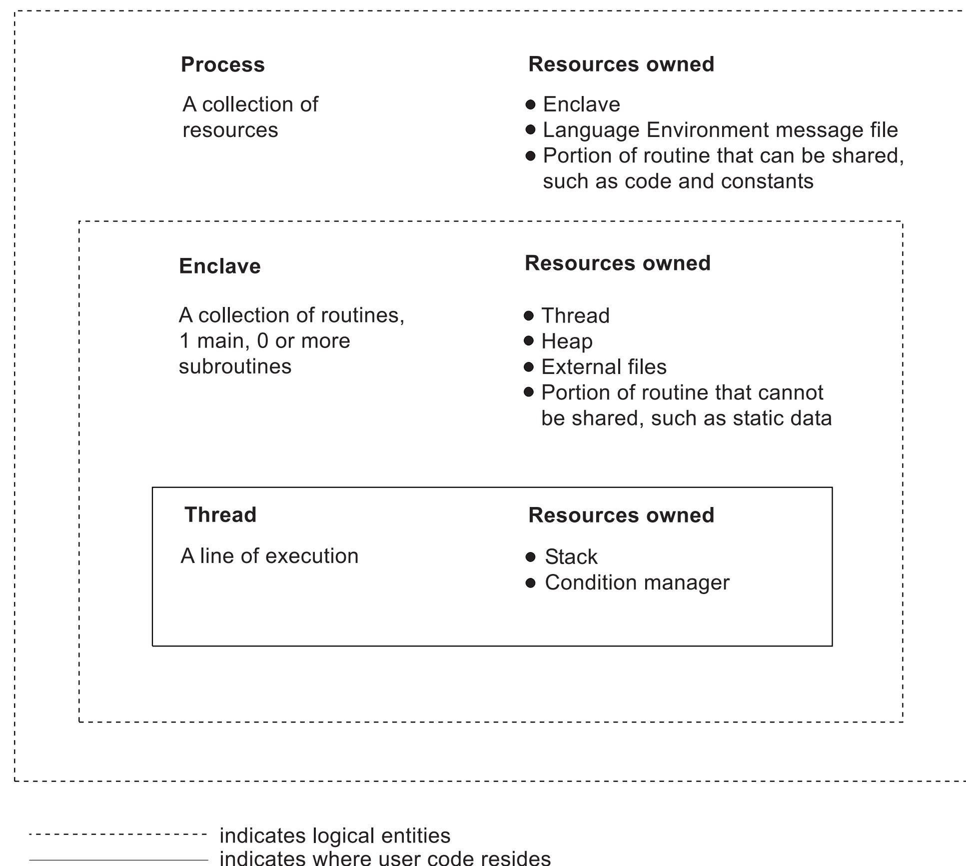


Figure 21. Program Management Model Illustration of Resource Ownership

Processes

A *process* is a collection of resources, both application code and data, consisting of one or more related enclaves (described in the next section). The process is the outermost or highest level run-time component of the common run-time environment. The resources maintained at the process level do not affect the language semantics of an application running at the enclave level.

The LE/VSE library is an example of the type of resource that is maintained at the process level. The LE/VSE library is loaded at process initialization, although it could be loaded for any of the individual enclaves within the process at enclave initialization. The process is used in the same way by all enclaves created within the process. It has no effect on the HLL semantics of applications running within each of the enclaves.

Each process has an address space that is logically separate from those of other processes. Except for communications with each other using certain LE/VSE mechanisms, no resources are shared between processes; processes do not share storage, for example. A process can create other processes. However, all processes are independent of one another; they are not hierarchically related.

LE/VSE Implementation Information

Although the Language Environment program model supports applications consisting of one or more processes, LE/VSE Version 1 Release 4 supports only a single process for each application that runs in the common run-time environment.

Enclaves

A key feature of the program management model is the *enclave*, which consists of one or more phases, each containing one or more separately compiled, bound routines. A phase can include HLL routines, assembler routines, and LE/VSE routines.

The Enclave Defines the Scope of Language Semantics

By definition, the scope of a language statement is that portion of code in which it has semantic effect. The enclave defines the scope of the language semantics for its component routines, just as a COBOL run unit defines the scope of semantics of a COBOL program. Scope encompasses names, external data sharing, and control statements such as C's `exit()`, COBOL's STOP RUN, and PL/I's STOP and EXIT statements.

The Enclave Defines the Scope of the Definition of the Main Routine and Subroutines: The enclave boundary defines whether a routine is a *main* routine or a *subroutine*. The first routine to run in the enclave is known as the main routine in LE/VSE. All others are designated subroutines of the main routine.

The first routine invoked in the enclave must be capable of being designated main according to the rules of the language of the routine. For example, a main routine in an LE/VSE-conforming PL/I application would be the PROC OPTIONS (MAIN) routine. All other routines invoked in the enclave must be capable of being a subroutine according to the rules of the languages of the routines.

If a routine is capable of being invoked as either a main or subroutine, and recursive invocations are allowed according to the rules of the language, the routine can be invoked multiple times within the enclave. The first of these invocations could be as a main routine and the others as subroutines.

The Enclave Defines the Scope and Visibility of the Following Types of Data:

- Automatic data: Automatic data is allocated with the same value on entry and reentry into a routine if it has been initialized to that value in the semantics of the language used, for example, data declared using the PL/I INIT() option. Values of the data at exit from the routine are not retained for the next entry into the routine. The scope of automatic data is a routine invocation within an enclave.
- External data: External data persists over the lifetime of an enclave and retains last-used values whenever a routine is reentered. The scope of external data is that of the enclosing enclave; all routines invoked within the enclave recognize the external data. Examples are C data objects of `extern` storage class, COBOL data items defined with the EXTERNAL attribute, and PL/I data declared as EXTERNAL.
- Local data: The scope of local data is that of the enclosing enclave; however, local data is recognized only by the routine that defines it. Examples are any C or PL/I variable with block scope and COBOL WORKING-STORAGE.

The Enclave Defines the Scope of Language Statements: The enclave defines the scope of language statements—for example, those that stop execution of the outermost routine within an enclave. C's `exit()`, COBOL's STOP RUN, and PL/I's STOP and EXIT statements are examples of such statements. When one of these statements is executed, the main routine within the enclave terminates. Thus, the enclave defines the scope of the language statements.

Prior to returning, resources obtained by the routines in the enclave are released and any open files (other than the LE/VSE message file) are closed.

Additional Enclave Characteristics

Management of Resources: The enclave manages most LE/VSE resources, such as the thread and heap storage, other than the message file (which is managed as a process-level resource). Heap storage, for example, is shared among all threads within an enclave. Allocated heap storage remains allocated until explicitly freed or until the enclave terminates. None of the enclave-managed resources is shared between enclaves.

LE/VSE Implementation Information

Multiple Enclaves: Although the Language Environment program management model supports multiple enclaves within a single process, LE/VSE Version 1 Release 4 provides explicit support for only a single enclave within a single process. Under some circumstances, however, multiple enclaves can exist within a single process. A description of how to create multiple, or nested, enclaves can be found in Chapter 28, "Using Nested Enclaves," on page 393.

Threads

Within each enclave is a *thread*, the basic run-time path represented by the machine state; conditions raised during execution are isolated to that run-time path.

Threads share all of the resources of an enclave and therefore do not need to selectively create or load new copies of resources, code, or data. Although a thread does not own its storage, it can address all storage within the enclave. All threads are independent of one another and are not related hierarchically. A thread is dispatched with its own run-time stack, instruction counter, registers, and condition handling mechanisms.

Because threads operate with unique run-time stacks, they can run concurrently within an enclave and allocate and free their own storage. Concurrent, or parallel, processing, is useful when code is event-driven, or for improving the performance of a large application.

LE/VSE Implementation Information

Multiple Threads: Although the full Language Environment program management model supports multiple threads within an enclave, LE/VSE Version 1 Release 4 supports only a single thread within an enclave.

The Full Language Environment Program Management Model

Figure 22 on page 80 illustrates the relationship between the various entities that make up the Language Environment program management model.

As Figure 22 on page 80 shows, each process consists of one or more enclaves. An enclave consists of one main routine with any number of subroutines. External data is available only within the enclave in which it resides. External data items that happen to be identically named in different enclaves reference distinct storage locations; the scope of external data, as described earlier, is the enclave. The threads can create enclaves, which can create more threads, and so on.

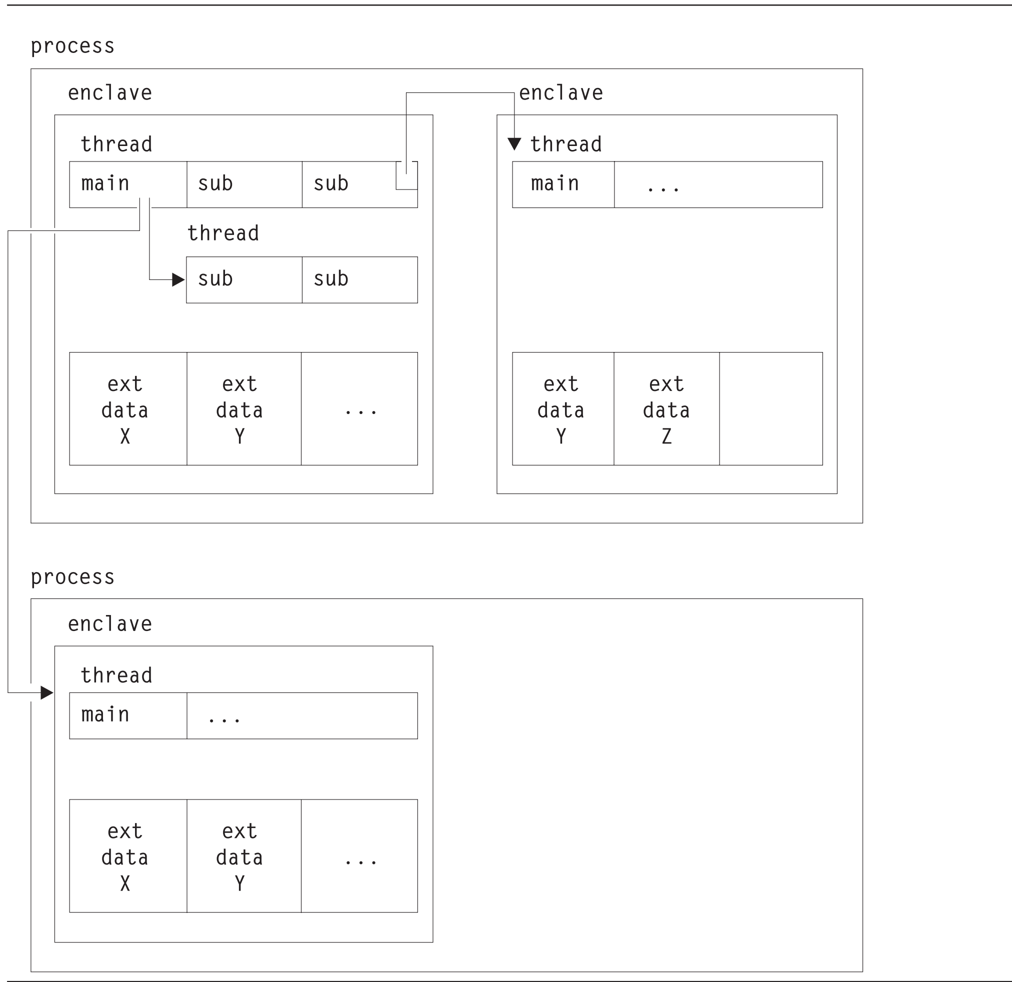


Figure 22. Overview of the Full Language Environment Program Management Model

Chapter 10. Stack and Heap Storage

LE/VSE provides services that control the stack and heap storage used at run time. LE/VSE-conforming HLLs and assembler routines use these services for all storage requests.

Understanding the Basics

LE/VSE provides the following types of storage:

Stack storage
: *Stack storage* is automatically created by LE/VSE and is used for routine linkage and automatic storage. Refer to "Stack Storage Overview" on page 83 for more information.

Heap storage
: *Heap storage* is dynamically allocated at a routine's first request for storage that has a lifetime not related to the execution of the current routine. You can control allocation and freeing of heap storage using LE/VSE callable services. Refer to "Heap Storage Overview" on page 85 for more information.

Table 25 summarizes the ways in which LE/VSE-conforming languages use stack and heap storage.

Table 25. Usage of Stack and Heap Storage by LE/VSE-Conforming Languages

Language	Stack	Heap
C	Automatic variables Library routines	Variable allocated by: `malloc` function `calloc` function `realloc` function Static external (RENT)
COBOL	Intrinsic functions Library routines	WORKING-STORAGE variables
PL/I	Automatic variables Library routines	BASED variables CONTROLLED variables AREA variables Variables allocated by ALLOCATE statement

The remainder of this section further discusses stack and heap storage concepts and terminology.

© Copyright IBM Corp. 1991, 2005

Related Options and Services

Run-Time Options

ANYHEAP
Allocates library (HLL and LE/VSE) heap storage above or below 16MB

BELOWHEAP
Allocates library heap storage below 16MB

HEAP Allocates storage for user-controlled dynamically allocated variables

LIBSTACK
Used by library routine stack frames that must be below 16MB

RPTSTG
Generates a storage report

STACK
Used by library routine stack frames that can reside anywhere in storage

STORAGE
Controls the initial content and amount of storage reserved for the out-of-storage condition

Callable Services

CEECRHP
Defines additional heaps

CEECZST
Changes the size of a previously allocated heap element

CEEDSHP
Discards an entire heap created with CEECRHP

CEEFRST
Frees storage allocated by CEEGTST or an intrinsic language function

CEEGTST
Gets storage from a heap whose ID you specify

CEE5RPH
Sets the heading displayed at the top of the storage options report

See *LE/VSE Programming Reference* for syntax information on run-time options and callable services.

Stack Storage Overview

Stack [2] storage [3] is the storage provided by LE/VSE that is needed for routine linkage and any automatic storage. It is allocated on entry to a routine or block, and freed on the subsequent return. It is a contiguous area of storage obtained directly from the operating system. Stack storage is automatically provided at thread initialization and is available in the *user stack* [2].

The user stack is used by both library routines and compiled code. Stack storage is also available in the *library stack*, which is an independent area of stack storage, allocated below the 16MB line, designed to be used only by library routines.

A *storage stack* is a data structure that supports procedure or block invocation (call and return). It is used to provide both the storage required for the application initialization and any automatic storage used by the called routine. Each thread has a separate and distinct stack.

The storage stack is divided into large segments of storage called *stack segments*, which are further divided into smaller segments called *stack frames* [3], also known as *dynamic storage areas* (*DSAs*). A stack frame, or DSA, is dynamically acquired storage composed of a register save area and an area available for dynamic storage allocation for items such as program variables. Stack frames are added to the user stack when a routine is entered, and removed upon exit in a last in, first out (LIFO) manner. Stack frame storage is acquired during the execution of a program and is allocated every time a procedure, function, or block is entered, as, for example, when a call is made to an LE/VSE callable service, and is freed when the procedure or block returns control.

The first segment used for stack storage is called the *initial stack segment*. When the initial stack segment becomes full, a second segment, or *stack increment* is obtained from the operating system. As each succeeding stack increment becomes full, another is obtained from the operating system as needed. The size of the initial stack segment and the size of the increments are specified by the *init_size* and *incr_size* parameters of the STACK run-time option. If the STACK option is not specified, LE/VSE uses the installation default or application default as the initial stack segment size (see *LE/VSE Programming Reference* for more information on using the STACK run-time option).

Figure 23 on page 84 shows the LE/VSE stack storage model.

---

2. The term stack, as used in this chapter refers to the *user stack*, which is an independent area of stack storage that can be located above or below the 16MB line, designed to be used by both library routines and compiled code.

3. All references to *stack storage* and *stack frame* in this chapter are to real storage allocation, as opposed to *invocation stack*, which refers to a conceptual stack.

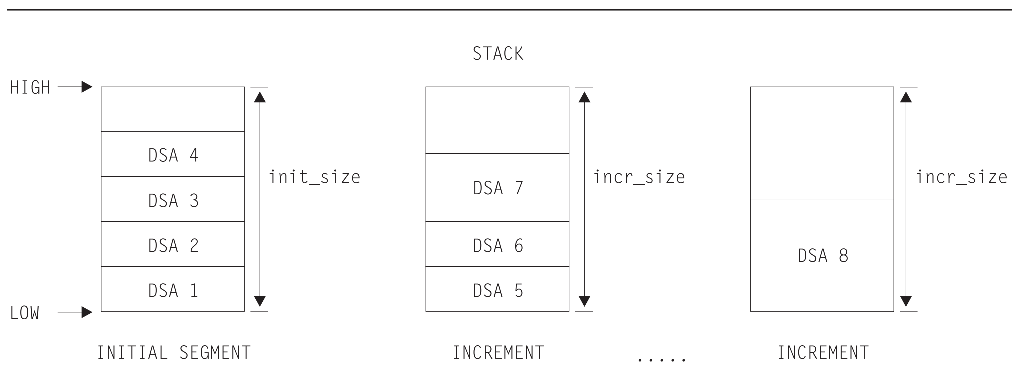


Figure 23. LE/VSE Stack Storage Model

Tuning Stack Storage

For best performance, the initial stack segment should be large enough to satisfy all requests for stack storage. The LE/VSE storage report generated by the RPTSTG(ON) option (see *LE/VSE Programming Reference*) shows you how much stack storage is being used, the total number of segments allocated to the stack, and the recommended values for the STACK run-time option. An initial stack segment that is too large can waste storage and degrade overall system performance, especially under CICS where storage is limited.

You can tune stack storage by using the LE/VSE STACK run-time option; see *LE/VSE Programming Reference* for syntax information.

Note: RPTSTG(ON), as well as the STORAGE run-time option (see *LE/VSE Programming Reference*) can have a negative impact upon the performance of your application, because as the application runs, statistics are kept on storage requests. Therefore, always specify the IBM-supplied default setting RPTSTG(OFF) when running production jobs. Use RPTSTG(ON) and STORAGE only when debugging and/or tuning applications.

COBOL Considerations

Ensure that your COBOL applications are not link-edited with the VS COBOL II space management tuning module, IGZETUN. IGZETUN is not supported by LE/VSE and causes an informational message to be logged. Logging this message for each application inhibits performance.

PL/I Storage Considerations

PL/I automatic storage is provided by the LE/VSE user stack. Automatic storage above the 16MB line is supported under control of the LE/VSE STACK run-time option. When the LE/VSE user stack is above 16MB, PL/I temporaries (dummy arguments) and parameter lists (for reentrant/recursive blocks) also reside above 16MB. As long as the application is AMODE(31), STACK(,,ANY) is supported. The stack frame size for an individual block is constrained to 16MB. This means the size of an automatic aggregate, temporary variable, or dummy argument cannot exceed 16MB.

Heap Storage Overview

Heap storage is used to allocate storage that has a lifetime not related to the execution of the current routine; it remains allocated until you explicitly free it or until the enclave terminates. You can control allocation and freeing of heap storage using LE/VSE callable services, and tune heap storage using the LE/VSE HEAP run-time option; consult *LE/VSE Programming Reference* for details.

Heap storage is shared among all program units and all threads in an enclave. Any thread can free heap storage. You can free one element at a time with the CEEFRST callable service, or you can free all heap elements at once using CEEDSHP. You cannot, however, discard the initial heap.

Storage can be allocated or freed with any of the HLL storage facilities, such as the C `malloc()` and `calloc()` functions or the PL/I ALLOCATE statement, along with the LE/VSE storage services. For HLLs with no intrinsic function for storage management, such as COBOL, you can use the LE/VSE storage services.

Heap storage (sometimes referred to simply as a 'heap') is a collection of one or more *heap segments* comprised of an *initial heap segment*, which is dynamically allocated at the first request for heap storage, and, as needed, one or more *heap increments*, allocated as additional storage is required. The initial heap is provided by LE/VSE and does not require a call to the CEECRHP service. The initial heap is identified by *heap_id=0*. It is also known as the *user heap*. See Figure 24 on page 86 for an illustration of LE/VSE heap storage.

Heap segments, which are contiguous areas of storage obtained directly from the operating system, are subdivided into individual *heap elements*. Heap elements are obtained by a call to the CEEGTST service, and are allocated within each segment of the initial heap by the LE/VSE storage management routines. When the initial heap segment becomes full, LE/VSE gets another segment, or increment, from the operating system.

The size of the initial heap segment is governed by the *init_size* parameter of the HEAP run-time option. (See *LE/VSE Programming Reference*.) The *incr_size* parameter governs the size of each heap increment.

A *named heap* is set up specifically by a call to the CEECRHP service, which returns an identifier when the heap is created. *Additional heaps* can also be created and controlled by calls to CEECRHP.

Additional heaps provide a means of isolating logical groups of data. Use additional heaps when you need to group storage objects together so they can be freed at once (with a single call to CEEDSHP), rather than freed one element at a time (with calls to CEEFRST).

Library routines occasionally use a heap called the *library heap* for storage below 16MB. The size of this heap is controlled by the BELOWHEAP run-time option. The library heap and the BELOWHEAP run-time option have no relation to heaps created by CEECRHP. If an application program creates a heap using CEECRHP, library routines never use that heap (except, of course, the storage management library routines CEEGTST, CEEFRST, CEECZST, and CEEDSHP). The library heap can be tuned with the BELOWHEAP run-time option.

Note: The LE/VSE anywhere heap and below heap are reserved for run-time library usage only. Application data and variables are not kept in these heaps. You normally should not adjust the size of these heaps unless the storage report indicates excessive segments allocated for the anywhere or below heaps, or if too much storage has been allocated.

You can use the LE/VSE STORAGE option to diagnose the use of uninitialized and freed storage.

See Chapter 5, "Using Run-Time Options," on page 33 and *LE/VSE Programming Reference* for more information on using LE/VSE run-time options.

Figure 24 shows the LE/VSE heap storage model.

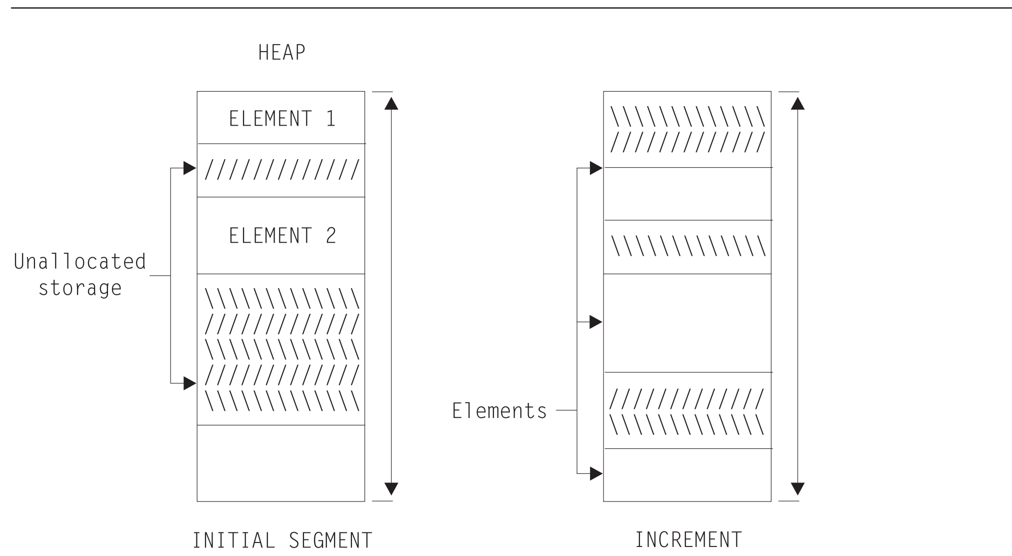


Figure 24. LE/VSE Heap Storage Model

Heap IDs Recognized by the LE/VSE Heap Manager

Table 26 lists LE/VSE heaps and their respective purposes.

Table 26. Heap IDs Recognized by LE/VSE Heap Manager

Heap Name	Heap ID	Intended Purpose	Created By	Disposed By
Initial heap User heap	0	Application program data. Common heap used by language intrinsic functions and COBOL WORKING-STORAGE data items. CEEDSHP has no effect on the initial heap. COBOL access is by LE/VSE callable services.	Enclave initialization. Size and location determined from HEAP run-time option.	Enclave termination
Additional heaps and user heap	(Returned by CEECRHP)	Collections of application program data that can be quickly disposed with a single CEEDSHP call.	Call to CEECRHP. Arguments define heap size, location, and other characteristics.	Call to CEEDSHP Enclave termination

AMODE Considerations for Heap Storage

The *initsz24* and *incrsz24* parameters of the HEAP run-time option control the initial size and subsequent increments of heap storage allocated below the 16MB line. This storage is required for AMODE(24) applications running with the ALL31(OFF) and HEAP(,,ANYWHERE) run-time options in effect.

For example, suppose the initial heap segment is allocated above 16MB. If an AMODE(24) routine requests storage from this initial heap, LE/VSE must allocate a heap segment from below the 16MB line so that the AMODE(24) routine can address the storage.

Tuning Heap Storage

For best performance, the initial heap segment should be large enough to satisfy all requests for heap storage. The LE/VSE storage report generated by the RPTSTG(ON) run-time option (see *LE/VSE Programming Reference*) shows you how much heap storage is being used, the total number of segments allocated to the heap, and the recommended values for the HEAP, ANYHEAP, and BELOWHEAP run-time options.

Note: RPTSTG(ON), as well as the STORAGE run-time option (see *LE/VSE Programming Reference*) can have a negative impact on the performance of your application. Therefore, always specify the IBM-supplied default setting RPTSTG(OFF) when running production jobs. Use RPTSTG(ON) and STORAGE(NONE,NONE,NONE) only to debug applications.

COBOL Considerations

Ensure that your COBOL applications are not link-edited with the VS COBOL II space management tuning module, IGZETUN. IGZETUN is not supported by LE/VSE and causes an informational message to be logged. Logging this message for each application inhibits performance.

Storage Performance Considerations

Use the RPTSTG(ON) option to generate a report about the amount of storage your application uses in various LE/VSE storage classes (such as STACK, HEAP, and LIBSTACK). You can also use the report to determine your application's minimum storage requirements and the number of segments allocated and freed. You can use this information to tune your application to minimize the number of segments allocated and freed. Before putting your application into production, be sure to specify the RPTSTG(OFF) option so that no storage report is generated. RPTSTG(ON) can have a negative impact on the performance of your application, because as the application runs, statistics are kept on storage requests.

COBOL and LE/VSE Storage Considerations

Table 27 on page 88 shows the interactions between using the ALL(31) run-time option and whether your COBOL routine is coded as reentrant or non-reentrant.

Table 27. COBOL Storage Usage

If you specify ... LE/VSE Run-Time Option	COBOL Compiler RENT Option	COBOL Compiler DATA Option	The effect on storage is ...
ALL31(ON)	RENT	DATA(31)	This usage scenario minimizes the amount of storage allocated below the 16MB line. Control Blocks: As many of the control blocks as possible are placed above the 16MB line in ANYHEAP storage. Working Storage (non-external data): allocated from HEAP storage above the 16MB line. Working Storage (external data): allocated from HEAP storage above the 16MB line.
ALL31(ON)	RENT	DATA(24)	Control Blocks: Same as above. Working Storage (non-external data): allocated from HEAP storage below the 16MB line. Working Storage (external data): allocated from HEAP storage below the 16MB line.
ALL31(ON)	NORENT[1]	DATA(31)	Control Blocks: Same as above. Working Storage (non-external data): is part of the application phase. HEAP storage is **not** allocated for this. Working Storage (external data): allocated from HEAP storage above the 16MB line.
ALL31(ON)	NORENT[1]	DATA(24)	Control Blocks: Same as above. Working Storage (non-external data): is part of the application phase. HEAP storage is **not** allocated for this. Working Storage (external data): allocated from HEAP storage below the 16MB line.
ALL31(OFF)	RENT	DATA(31)	Control Blocks: Some control blocks allocated in ANYHEAP when ALL31 is ON are allocated in BELOWHEAP. Working Storage (non-external data): allocated from HEAP storage above the 16MB line. Working Storage (external data): allocated from HEAP storage below the 16MB line.
ALL31(OFF)	RENT	DATA(24)	Control Blocks: Same as above. Working Storage (non-external data): allocated from HEAP storage below the 16MB line. Working Storage (external data): allocated from HEAP storage below the 16MB line.

Table 27. COBOL Storage Usage (continued)

If you specify ... LE/VSE Run-Time Option	COBOL Compiler RENT Option	COBOL Compiler DATA Option	The effect on storage is ...
ALL31(OFF)	NORENT	DATA(31)	Control Blocks: Same as above. Working Storage (non-external data): is part of the application phase. HEAP storage is **not** allocated for this. Working Storage (external data): allocated from HEAP storage above the 16MB line.
ALL31(OFF)	NORENT	DATA(24)	Control Blocks: Same as above. Working Storage (non-external data): is part of the application phase. HEAP storage is **not** allocated for this. Working Storage (external data): allocated from HEAP storage below the 16MB line.

Note:

1. When you link-edit a COBOL program compiled with the NORENT compiler option, the default addressing mode of the link-edited phase is AMODE(ANY). This might result in your program being invoked in 24-bit addressing mode. In order to specify the LE/VSE ALL31(ON) run-time option, your program must be invoked in 31-bit addressing mode. Therefore, you should link-edit your application as AMODE(31). You can use the MODE linkage editor control statement to override the default addressing mode.

Dynamic Storage Services

LE/VSE provides callable services that let you get and free heap storage at selected points in your application. Stack storage is automatically allocated upon entry into a routine and freed upon exit, but you must allocate heap storage, which persists until you free it or until your application terminates.

Each time your application runs, the setting of the HEAP run-time option specifies the size of an initial heap from which heap storage is allocated. You can allocate storage out of this initial heap whenever your application requires it. Call CEEGTST (Get Heap Storage) and specify an ID identifying the initial heap and the portion of storage in the initial heap that you require. When your application no longer requires the storage, you can call the CEEFRST (Free Heap Storage) service with the address of the element to free it.

CEECRHP (Create New Additional Heap) allows you to identify a heap, other than the initial heap, from which to get and free storage. You can use CEEGTST to allocate elements from the newly created heap. One advantage of this approach is that CEECRHP allows you to group storage elements together and to use CEEDSHP (Discard Heap) to discard them all at once when you no longer need them.

For a description and syntax of each LE/VSE dynamic storage callable service, see *LE/VSE Programming Reference*.

Examples of Callable Storage Services

This section contains examples that use callable services. The first group of examples use CEEGTST and CEEFRST to build a linked list. The second group of examples use CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST, and CEEDSHP to manage storage.

C Example of Building a Linked List

The following is an example of how to build a linked list in a C program using callable services.

```
/*Module/File Name: EDCLLST   */
 /****************************************************************
  **FUNCTION      : CEEGTST - obtain storage from user heap     *
  **                        for a linked list.                 *
  **              : CEEFRST - free linked list storage         *
  **                                                           *
  **   This example illustrates the construction of a linked   *
  **   list using the LE/VSE storage management services.      *
  **                                                           *
  **   1.  Storage for each list element is allocated from the *
  **       user heap,                                          *
  **   2.  The list element is initialized and appended to the *
  **       list.                                               *
  **   3.  After three members are appended, the list traversed *
  **       and the count saved in each element is displayed.   *
  **   4.  The link list storage is freed.                     *
  **                                                           *
  ****************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>
void main ()
{
  _INT4 HEAPID;
  _INT4 HPSIZE;
  _INT4 LCOUNT;
  _FEEDBACK FC;
  _POINTER ADDRSS;
  struct LIST_ITEM
       { _INT4 COUNT;
         struct LIST_ITEM *NEXT_ITEM;
       } ;
  struct LIST_ITEM *ANCHOR;
  struct LIST_ITEM *CURRENT;
  _INT4 NBYTES = sizeof(struct LIST_ITEM);

  printf ( "\n**********************************\n");
  printf ( "\nCESCSTO C Example is now in motion\n");
  printf ( "\n**********************************\n");
  ANCHOR = NULL;

  for ( LCOUNT = 1; LCOUNT < 4; LCOUNT++)
    {
```

Figure 25. C Example Using CEEGTST and CEEFRST to Build a Linked List (Part 1 of 2)

```
    /*******************************************************
     * Call CEEGTST to get storage from user heap         *
     *******************************************************/
    CEEGTST ( &HEAPID , &NBYTES , &ADDRSS , &FC );
    if ( (_FBCHECK (FC , CEE000) == 0) && ADDRSS != 0 )
      /*  **********************************************
       *  If storage is gotten successfully, the linked  *
       *  list elements are pointed to by the pointer    *
       *  variable CURRENT. Append element to the end of *
       *  the list. The list origin is pointed to by the *
       *  variable ANCHOR.                               *
       *  **********************************************/          {
        if (ANCHOR == NULL)
          {
            ANCHOR =(struct LIST_ITEM *) ADDRSS;
          }else{
            CURRENT -> NEXT_ITEM =(struct LIST_ITEM *)ADDRSS;
          }
          CURRENT =(struct LIST_ITEM *) ADDRSS;
          CURRENT -> NEXT_ITEM = NULL;
          CURRENT -> COUNT = LCOUNT;
      }else{
        printf ( "Error in getting user storage\n" );
      }
    }
    /*********************************************************
     *  On completion of the above loop, we have the        *
     *  following layout:                                   *
     *                                                      *
     *  ANCHOR  -->  LIST-ITEM1 --> LIST-ITEM2 --> LIST-ITEM3*
     *                                                      *
     *  Loop thru list items 1 thru 3 and print out the     *
     *  identifying item number saved in the COUNT field.   *
     *                                                      *
     *  Test the LCOUNT variable to verify that three items *
     *  were indeed in the linked list.                     *
     *******************************************************/
 CURRENT = ANCHOR;
 while (CURRENT)
   {
     printf("This is list item %d\n", CURRENT->COUNT) ;
     ADDRSS = CURRENT;
     LCOUNT = CURRENT -> COUNT;
     CURRENT = CURRENT -> NEXT_ITEM;
     /****************************************************
      * Call CEEFRST to free this piece of storage         *
      ****************************************************/
     CEEFRST ( &ADDRSS , &FC );
     if ( _FBCHECK (FC , CEE000) == 0 )
       {
       }else{
          printf ( "Error freeing storage from heap\n" );
       }
   }
  if (LCOUNT == 3)
   {
    printf ( "\n********************************\n" );
    printf ( "\nC linked list example ended.\n" );
    printf ( "\n********************************\n" );
    exit(0);
   }else{
    printf ( "Error in constructing linked list\n" );
   }
 exit(-1);
}
```

Figure 25. C Example Using CEEGTST and CEEFRST to Build a Linked List (Part 2 of 2)

COBOL Example of Building a Linked List

The following is an example of how to build a linked list in a COBOL program using callable services.

```
CBL C,LIB,RENT,LIST,APOST
      *Module/File Name: IGZTLLST
      ******************************************************************************
      ** CESCSTO - Drive CEEGTST - obtain storage from user heap for a linked list. *
      **           and   CEEFRST - free linked list storage                         *
      **                                                                            *
      **   This example illustrates the construction of a linked list using the    *
      **   LE storage management services.                                          *
      **                                                                            *
      **   1.  Storage for each list element is allocated from the user heap,       *
      **                                                                            *
      **   2.  The list element is initialized and appended to the list.            *
      **                                                                            *
      **   3.  After three members are appended, the list traversed and the data    *
      **       saved in each element is displayed.                                  *
      **                                                                            *
      **   4.  The link list storage is freed.                                      *
      ******************************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CESCSTO.
       DATA DIVISION.
      ********************************************************
      ** Storage management parameters, including pointers **
      ** for the returned storage addresses.              **
      ********************************************************
       WORKING-STORAGE SECTION.
       01  LCOUNT                 PIC 9 USAGE DISPLAY VALUE 0.
       01  HEAPID                 PIC S9(9) BINARY VALUE 0.
       01  NBYTES                 PIC S9(9) BINARY.
       01  FC.
           05  FILLER             PIC X(8).
           COPY CEEIGZCT.
           05  FILLER             PIC X(4).
       01  ADDRSS USAGE IS POINTER VALUE NULL.
       01  ANCHOR USAGE IS POINTER VALUE NULL.
      ********************************************************
      ** Define variables in linkage section in order to   **
      ** reference storage returned as addresses in        **
      ** pointer variables by Language Environment.        **
      ********************************************************
       LINKAGE SECTION.
       01  LIST-ITEM.
           05  CHARDATA           PIC X(80) USAGE DISPLAY.
           05  NEXT-ITEM USAGE IS  POINTER.
       PROCEDURE DIVISION.
       0001-BEGIN-PROCESSING.
           DISPLAY '**************************************'.
           DISPLAY 'CESCSTO COBOL Example is now in motion. '.
           DISPLAY '**************************************'.

            MOVE LENGTH OF LIST-ITEM TO NBYTES
           PERFORM 3 TIMES
             ADD 1 TO LCOUNT
             CALL 'CEEGTST' USING HEAPID , NBYTES,
                           ADDRSS , FC
```

Figure 26. COBOL Example Using CEEGTST and CEEFRST to Build a Linked List (Part 1 of 2)

```
*******************************************************************************
** If storage is gotten successfully, an address is returned by LE in the     *
** ADDRSS parameter. The address of variable LIST-ITEM in the linkage section *
** can now be SET to address the aquired storage. LIST-ITEM is appended to the*
** end of the list. The list origin is pointed to by the variable ANCHOR.     *
*******************************************************************************
       IF CEE000 THEN
           IF ANCHOR = NULL THEN
               SET ANCHOR TO ADDRSS
           ELSE
               SET NEXT-ITEM TO ADDRSS
           END-IF
           SET ADDRESS OF LIST-ITEM TO ADDRSS
           SET NEXT-ITEM TO NULL
           MOVE ' ' TO CHARDATA
           STRING 'This is list item number ' LCOUNT
               DELIMITED BY SIZE INTO CHARDATA
       ELSE
           DISPLAY 'Error in obtaining storage from heap'
           GOBACK
       END-IF
     END-PERFORM.
*********************************************************
** On completion of the above loop, we have the       **
** following layout:                                  **
**                                                    **
** ANCHOR --> LIST-ITEM1 --> LIST-ITEM2 --> LIST-ITEM3 **
**                                                    **
** Loop thru list items 1 thru 3 and print out the    **
** identifying text written in the CHARDATA fields.   **
**                                                    **
** Test a counter variable to verify that three items  **
** were indeed in the linked list.                    **
*********************************************************
     MOVE 0 TO LCOUNT.
     PERFORM WITH TEST AFTER UNTIL (ANCHOR = NULL)
       SET ADDRESS OF LIST-ITEM TO ANCHOR
       DISPLAY CHARDATA
       SET ADDRSS TO ANCHOR
       SET ANCHOR TO NEXT-ITEM
       PERFORM 100-FREESTOR
       ADD 1 TO LCOUNT
     END-PERFORM.
     IF (LCOUNT = 3 )
       THEN
         DISPLAY '***************************************'
         DISPLAY 'CESCSTO COBOL Example is now ended.   '
         DISPLAY '***************************************'
       ELSE
         DISPLAY 'Error in List construction .'
     END-IF.

     GOBACK.

 100-FREESTOR.
*********************************************************
*     Call CEEFRST to free this storage from user heap **
*********************************************************
     CALL 'CEEFRST' USING ADDRSS , FC.
     IF CEE000 THEN
         NEXT SENTENCE
     ELSE
         DISPLAY 'Error freeing storage from heap'
     END-IF.
```

Figure 26. COBOL Example Using CEEGTST and CEEFRST to Build a Linked List (Part 2 of 2)

PL/I Example of Building a Linked List

The following is an example of how to build a linked list in a PL/I program using callable services.

```
*PROCESS MACRO;
*Process lc(101),opt(0),s,map,list,stmt,a(f),ag        ;
   /***************************************************/
   /*Module/File Name: IBMLLST                        */
   /***************************************************/
   /**                                               **/
   /** FUNCTION : CEEGTST - obtain storage from user  **/
   /**                      heap for a linked list.   **/
   /**          : CEEFRST - free linked list storage  **/
   /**                                               **/
   /**   This example illustrates the construction of **/
   /**   a linked list using the LE/VSE storage      **/
   /**   management services.                        **/
   /**                                               **/
   /**   1.  Storage for each list element is        **/
   /**       allocated from the user heap,           **/
   /**                                               **/
   /**   2.  The list element is initialized and     **/
   /**       appended to the list.                   **/
   /**                                               **/
   /**   3.  After three members are appended, the   **/
   /**       list traversed and the data saved in    **/
   /**       each element is displayed.              **/
   /**                                               **/
   /**   4.  The link list storage is freed.         **/
   /**                                               **/
   /***************************************************/
 CESCSTO: PROC OPTIONS(MAIN);

   %INCLUDE  CEEIBMAW;
   %INCLUDE  CEEIBMCT;
   /***************************************************/
   DCL NULL BUILTIN;
   /***************************************************/
   /*  Storage management parameters, including       */
   /*  pointers for the returned storage addresses.   */
   /***************************************************/
   DCL HEAPID INT4 INIT (0);  /* heap ID for user heap */
   DCL NBYTES INT4            /* size of required heap */
              INIT (STORAGE(LIST_ITEM));
   DCL 01 FC  FEEDBACK;
   DCL ADDRSS POINTER,        /* Address of storage    */
       PREV   POINTER;        /* Address of prior item */
   DCL ANCHOR POINTER;        /* Link list anchor      */
   /***************************************************/
   /*  Declare linked list item as based structure.   */
   /***************************************************/
   DCL 01 LIST_ITEM BASED(ADDRSS), /* Map of list item */
          02  CHARDATA CHAR(80),
          02  NEXT_ITEM POINTER;

   PUT SKIP LIST('*****************************************');
   PUT SKIP LIST('PL/I linked list example is now in motion');
   PUT SKIP LIST('*****************************************');
   ANCHOR = NULL;
   DO LCOUNT = 1 TO 3;
```

Figure 27. PL/I Example Using CEEGTST and CEEFRST to Build a Linked List (Part 1 of 2)

```
      /****************************************************/
      /*   Call CEEGTST to get storage from user heap     */
      /****************************************************/
      CALL CEEGTST ( HEAPID, NBYTES, ADDRSS, FC );
      IF  FBCHECK( FC, CEE000)  THEN  DO;

         /*********************************************/
         /* If storage is obtained successfully, the   */
         /* linked list elements are based on the      */
         /* address of the storage obtained.  Append   */
         /* element to end of list. The  list origin   */
         /* is pointed to by the variable ANCHOR.      */
         /*********************************************/
         IF ( ANCHOR = NULL )  THEN
             ANCHOR = ADDRSS;
         ELSE
            PREV ->NEXT_ITEM = ADDRSS;
         NEXT_ITEM = NULL;
         CHARDATA = 'This is list item number ' || LCOUNT;
         PREV = ADDRSS;
         END;
      ELSE  DO;
         PUT SKIP LIST ( 'Error ' || FC.MsgNo
            || ' in getting user storage' );
         STOP;
         END;
      END;

   /****************************************************/
   /*  On completion of the above loop, we have the     */
   /*  following layout:                                */
   /*                                                   */
   /*  ANCHOR -> LIST_ITEM1 -> LIST_ITEM2 -> LIST_ITEM3 */
   /*                                                   */
   /*  Loop thru list items 1 thru 3 and print out the  */
   /*  identifying text written in the CHARDATA fields. */
   /*                                                   */
   /*  Test a counter variable to verify that three     */
   /*  items were indeed in the linked-list.            */
   /****************************************************/
   ADDRSS = ANCHOR;
   LCOUNT = 0;
   DO UNTIL (ADDRSS = NULL);

      PUT SKIP LIST(CHARDATA);
      /***********************************************/
      /*   Call CEEFRST to free this piece of storage  */
      /***********************************************/
      CALL CEEFRST ( ADDRSS, FC );
      IF  FBCHECK( FC, CEE000)  THEN  DO;
         LCOUNT = LCOUNT + 1;
         END;
      ELSE  DO;
         PUT SKIP LIST ( 'Error' || FC.MsgNo
            || ' freeing storage from heap');
         STOP;
         END;
      ADDRSS = NEXT_ITEM;
      END;

   IF  LCOUNT = 3  THEN  DO;
      PUT SKIP LIST('**************************************');
      PUT SKIP LIST('PL/I linked list example is now ended.');
      PUT SKIP LIST('**************************************');
      END;

END CESCSTO;
```

Figure 27. PL/I Example Using CEEGTST and CEEFRST to Build a Linked List (Part 2 of 2)

C Example of Storage Management

The following is an example of how to manage storage for a C program using callable services.

```
/*Module/File Name: EDCSTOR                                      */
/****************************************************************/
/*                                                              */
/* Function   : CEE5RPH - Set report heading                    */
/*            : CEECRHP - Create user heap                      */
/*            : CEEGTST - Obtain storage from user heap         */
/*            : CEECZST - Change size of this piece of storage  */
/*            : CEEFRST - Free this piece of storage            */
/*            : CEEDSHP - Discard user heap                     */
/*                                                              */
/*    This example illustrates the invocation of the LE/VSE     */
/*    Dynamic Storage Callable Services from a C program.       */
/*                                                              */
/*    1.  A report heading is set for display at the beginning  */
/*        of the storage or options report.                     */
/*                                                              */
/*    2.  A user heap is created.                               */
/*                                                              */
/*    3.  Storage is allocated from the user heap.              */
/*                                                              */
/*    4.  A change is made to the size of the allocated storage.*/
/*                                                              */
/*    5.  The allocated storage is freed.                       */
/*                                                              */
/*    6.  The user heap is discarded.                           */
/*                                                              */
/*****************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>
void main ()
{
  _CHAR80 RPTHEAD;
  _INT4 HEAPID;
  _INT4 HPSIZE;
  _INT4 NBYTES;
  _INT4 INCR;
  _INT4 OPTS;
  _INT4 STORALC;
  _POINTER ADDRSS;
  _FEEDBACK FC;
  printf ( "\n*********************************\n");
  printf ( "\nCE90STO C Example is now in motion\n");
  printf ( "\n*********************************\n");
  memset ( RPTHEAD , ' ' , 80 );
  memcpy ( RPTHEAD , "User defined report heading" , 27 );
  /*****************************************************
   * Call CEE5RPH to set the user defined report heading *
   *****************************************************/
  CEE5RPH ( RPTHEAD , &FC );
  if ( _FBCHECK ( FC , CEE000 ) != 0 )
      printf ( "Error in setting report heading\n" );
```

Figure 28. C Example Illustrating Calls to CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST, and CEEDSHP (Part 1 of 2)

```
  /******************************************************
   * Call CEECRHP to create a user heap                *
   ******************************************************/
  HEAPID = 0;
  HPSIZE = 1;
  INCR = 0;
  OPTS = 0;
  STORALC = 0;
  CEECRHP ( &HEAPID , &HPSIZE , &INCR , &OPTS , &FC );
  if ( _FBCHECK ( FC , CEE000 ) == 0 )
    {
    /******************************************************
     * Call CEEGTST to get storage from user heap         *
     ******************************************************/
    NBYTES = 4000;
    CEEGTST ( &HEAPID , &NBYTES , &ADDRSS , &FC );
    if ( ( _FBCHECK ( FC , CEE000 ) == 0 ) && ADDRSS != 0 )
      {
      /******************************************************
       * Call CEECZST to change size of heap element        *
       ******************************************************/
      NBYTES = 2000;
      CEECZST ( &ADDRSS , &NBYTES , &FC );
      if ( _FBCHECK ( FC , CEE000 ) == 0 )
        {
          STORALC = 1;
        }else{
          printf ( "Error in changing size of storage\n");
      }
      }else{
        printf ( "Error in getting user storage\n" );
    }
    }else{
      printf ( "Error in creating user heap\n" );
  }
 if (STORALC != 0)
  {
  /******************************************************
   * Call CEEFRST to free this piece of storage         *
   ******************************************************/
  CEEFRST ( &ADDRSS , &FC );

  if ( _FBCHECK ( FC , CEE000 ) == 0 )
    {
    /******************************************************
     * Call CEEDSHP to discard user heap                  *
     ******************************************************/
     CEEDSHP ( &HEAPID , &FC );
     if ( _FBCHECK ( FC , CEE000 ) == 0 )
       {
         printf ( "C Storage Example ended\n" );
         exit(0);
       }else{
         printf ( "Error discarding user heap\n" );
     }
     }else{
       printf ( "Error freeing storage from heap\n" );
    }
  }
  exit(-1);
}
```

Figure 28. C Example Illustrating Calls to CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST, and CEEDSHP (Part 2 of 2)

COBOL Example of Storage Management

The following is an example of how to manage storage for a COBOL program using callable services.

```
CBL LIB,APOST
      *Module/File Name: IGZTSTOR
      *************************************************************
      ** CE90STO - Call the following LE services:
      **           : CEE5RPH - Set report heading
      **           : CEECRHP - Create user heap
      **           : CEEGTST - obtain storage from user heap
      **           : CEECZST - change size of this piece of storage
      **           : CEEFRST - free this piece of storage
      **           : CEEDSHP - discard user heap
      **   This example illustrates the invocation of the LE
      **   Dynamic Storage Callable Services from a COBOL program.
      **
      **   1.  A report heading is set for display at the beginning
      **       of the storage or options report.
      **
      **   2.  A user heap is created.
      **
      **   3.  Storage is allocated from the user heap.
      **
      **   4.  A change is made to the size of the allocated storage.
      **
      **   5.  The allocated storage is freed.
      **
      **   6.  The user heap is discarded.
      *************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CE90STO.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  RPTHEAD                PIC X(80).
       01  HEAPID                 PIC S9(9) BINARY.
       01  HPSIZE                 PIC S9(9) BINARY.
       01  INCR                   PIC S9(9) BINARY.
       01  OPTS                   PIC S9(9) BINARY.
       01  ADDRSS                 USAGE IS POINTER.
       01  NBYTES                 PIC S9(9) BINARY.
       01  NEWSIZE                PIC S9(9) BINARY.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info           PIC S9(9) BINARY.
       PROCEDURE DIVISION.
       0001-BEGIN-PROCESSING.
           DISPLAY '*************************************'.
           DISPLAY 'CE90STO COBOL Example is now in motion. '.
           DISPLAY '*************************************'.
           MOVE 'User defined report heading' TO RPTHEAD.
```

Figure 29. COBOL Example Illustrating Calls to CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST and CEEDSHP (Part 1 of 2)

```
*************************************************************
*   Call CEE5RPH to set the user defined report heading
*************************************************************
     CALL 'CEE5RPH' USING RPTHEAD, FC.
     IF NOT CEE000 THEN
         DISPLAY 'Error in setting Report Heading'
         GOBACK
     END-IF.
*************************************************************
*   Call CEECRHP to create a user heap
*************************************************************
     MOVE 0 TO HEAPID.
     MOVE 1 TO HPSIZE.
     MOVE 0 TO INCR.
     MOVE 0 TO OPTS.
     CALL 'CEECRHP' USING HEAPID, HPSIZE, INCR, OPTS, FC.
     IF CEE000 of FC THEN
*************************************************************
*        Call CEEGTST to get storage from user heap
*************************************************************
         MOVE 4000 TO NBYTES
         CALL 'CEEGTST' USING HEAPID, NBYTES, ADDRSS, FC
         IF CEE000 of FC THEN
*************************************************************
*            Call CEECZST to change the size of heap element
*************************************************************
             MOVE 2000 TO NEWSIZE
             CALL 'CEECZST' USING ADDRSS, NEWSIZE, FC
             IF CEE000 of FC THEN
                 PERFORM 100-FREE-ALL
                 DISPLAY 'COBOL Storage example pgm ended'
             ELSE
                 DISPLAY 'Error in changing size of storage'
             END-IF
         ELSE
             DISPLAY 'Error in obtaining storage from heap'
         END-IF
     ELSE
         DISPLAY 'Error in creating user heap'
     END-IF.

     GOBACK.

 100-FREE-ALL.
*************************************************************
*     Call CEEFRST to free this storage from user heap
*************************************************************
     CALL 'CEEFRST' USING ADDRSS, FC.
     IF CEE000 of FC THEN
*************************************************************
*        Call CEEDSHP to discard user heap
*************************************************************
         CALL 'CEEDSHP' USING HEAPID, FC
         IF CEE000 THEN
             NEXT SENTENCE
         ELSE
             DISPLAY 'Error discarding user heap'
         END-IF
     ELSE
         DISPLAY 'Error freeing storage from heap'
     END-IF.
```

Figure 29. COBOL Example Illustrating Calls to CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST and CEEDSHP (Part 2 of 2)

PL/I Example of Storage Management

The following is an example of how to manage storage for a PL/I program using callable services.

```
*PROCESS MACRO;
 /*******************************************************/
 /*Module/File Name: IBMSTOR                            */
 /*******************************************************/
 /* FUNCTION : CEE5RPH - set report heading             */
 /*          : CEECRHP - create user heap               */
 /*          : CEEGTST - obtain storage from user heap  */
 /*          : CEEZCST - change size of storage block   */
 /*          : CEEFRST - free this piece of storage     */
 /*          : CEEDSHP - discard user heap              */
 /*  This example illustrates the use of the LE/VSE     */
 /*  Storage Callable Services from a PL/I program.     */
 /*                                                     */
 /*  1.  A report heading is set for display at the     */
 /*      beginning of the storage or options report.    */
 /*                                                     */
 /*  2.  A user heap is created.                        */
 /*                                                     */
 /*  3.  Storage is allocated from the user heap.       */
 /*                                                     */
 /*  4.  The size of allocated storage is changed.      */
 /*                                                     */
 /*  5.  The allocated storage is freed.                */
 /*                                                     */
 /*  6.  The user heap is discarded.                    */
 /*                                                     */
 /*******************************************************/

 CE90STO: PROC OPTIONS(MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL NULL    BUILTIN;
    DCL ADDRESS PTR  INIT(NULL); /* ADDRESS OF STORAGE    */
    DCL NEWSIZE INT4 INIT(2000); /* NEW STORAGE SIZE      */
    DCL RPTHEAD CHAR(80)
                     INIT('USER DEFINED REPORT HEADING');
    DCL HEAPID  INT4 INIT(0);    /* HEAP ID FOR CEECRHP   */
    DCL HPSIZE  INT4 INIT(1);    /* HEAP SIZE FOR CEECRHP */
    DCL INCR    INT4 INIT(0);    /* HEAP INCREMENT        */
    DCL NBYTES  INT4 INIT(4000); /* SIZE OF REQUIRED HEAP */
    DCL 01 FC   FEEDBACK;
    DCL OPTS    INT4 INIT(0);    /* HEAP OPTIONS          */

    PUT SKIP LIST('PL/I Storage example is now in motion');
    /**************************************************/
    /* Call CEE5RPH to set user defined report heading  */
    /**************************************************/
    CALL CEE5RPH ( RPTHEAD, FC );
    IF  &#170; FBCHECK( FC, CEE000)  THEN  DO;
       PUT SKIP LIST ( 'Error ' || FC.MsgNo
          || ' in setting Report Heading');
       STOP;
       END;
```

Figure 30. PL/I Example Illustrating Calls to CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST and CEEDSHP (Part 1 of 2)

```
   /****************************************************/
   /* Call CEECRHP to create user heap                 */
   /****************************************************/
   CALL CEECRHP ( HEAPID, HPSIZE, INCR, OPTS, FC );
   IF  FBCHECK( FC, CEE000)  THEN  DO;
      /*************************************************/
      /* Call CEEGTST to get storage from user heap    */
      /*************************************************/
      CALL CEEGTST ( HEAPID, NBYTES, ADDRESS, FC );
      IF  FBCHECK( FC, CEE000)  THEN  DO;
         /**********************************************/
         /* Call CEECZST to change the size of block   */
         /**********************************************/
         CALL CEECZST ( ADDRESS, NEWSIZE, FC );
         IF  FBCHECK( FC, CEE000)  THEN  DO;
            CALL FREE_ALL;
            PUT SKIP LIST ( 'PL/I Storage Example program ended');
            END;
         ELSE  DO;
            PUT SKIP LIST('Error ' || FC.MsgNo
               || ' in changing size of storage');
            STOP;
            END;
         END;
      ELSE  DO;
         PUT SKIP LIST( 'Error ' || FC.MsgNo
            || ' in getting user storage' );
         STOP;
         END;
      END;
   ELSE  DO;
      PUT SKIP LIST ('Error' || FC.MsgNo
         || ' in creating user heap');
      STOP;
      END;

   /* Logical end of Main program CE90STO */

FREE_ALL:  PROC;

   /****************************************************/
   /* Call CEEFRST to free this piece of storage       */
   /****************************************************/
   CALL CEEFRST ( ADDRESS, FC );
   IF  FBCHECK( FC, CEE000)  THEN  DO;
      /*************************************************/
      /*   Call CEEDSHP to discard user heap           */
      /*************************************************/
      CALL CEEDSHP ( HEAPID, FC );
      IF  &#170; FBCHECK( FC, CEE000)  THEN  DO;
         PUT SKIP LIST ( 'Error ' || FC.MsgNo
            || ' discarding user heap');
         STOP;
         END;
      END;
   ELSE  DO;
      PUT SKIP LIST ( 'Error ' || FC.MsgNo
         || ' freeing storage from heap');
      STOP;
      END;

END FREE_ALL;

END CE90STO;
```

Figure 30. PL/I Example Illustrating Calls to CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST and CEEDSHP (Part 2 of 2)

Chapter 11. LE/VSE Condition Handling Introduction

This chapter outlines the LE/VSE condition handling model. It describes what constitutes a condition in LE/VSE and how LE/VSE supplements existing HLL condition handling methods. It also presents several condition handling scenarios to demonstrate how LE/VSE condition handling works.

If you use mixed-language applications, it is especially important for you to know how LE/VSE condition handling works with existing high-level language (HLL) condition handling schemes.

The chapters that follow:

- Chapter 12, "LE/VSE and HLL Condition Handling Interactions," on page 121
- Chapter 13, "Coding a User-Written Condition Handler," on page 137
- Chapter 14, "Using Condition Tokens," on page 171

describe in detail the steps involved in condition handling under LE/VSE, HLL-specific condition handling considerations, and how you can communicate events that happen in a routine to another routine.

If your application is running under CICS, you should refer to the CICS-specific condition handling information, which is discussed in "Condition Handling under CICS" on page 305. If your application is running under DL/I, you should refer to the DL/I-specific condition handling information, which is discussed in "Condition Handling with DL/I" on page 314.

Understanding the Basics

The two main concepts of LE/VSE condition handling are its stack frame-based model and the unique, 12-byte condition token that it provides to communicate information about conditions to LE/VSE resources and services.

LE/VSE uses stack frames to keep track of a routine's order of execution, and the condition handlers available for each routine. This ensures that conditions can be isolated and handled precisely where they occur in a routine.

One of the most useful features of the condition handling model is the condition token: a 12-byte data type that contains information about each condition. You can use the condition token as a feedback code or to communicate with LE/VSE message services. Unlike a return code, which is specific to the caller and callee of a routine, a condition token communicates between all the routines involved in an application. A condition token contains more instance-specific information about a condition than does a return code.

LE/VSE supplements, but does not replace, existing HLL condition handling techniques such as C *signal handlers* (created using the `signal()` function), PL/I ON-units, and return code-based programming techniques. HLL condition handling techniques are discussed in Chapter 12, "LE/VSE and HLL Condition Handling Interactions," on page 121.

LE/VSE condition handling is most beneficial when used as part of mixed-language applications because it is consistent for all applications. If you are

© Copyright IBM Corp. 1991, 2005

coding in a single language, you can use the condition handling semantics of that language, but if you have any ILC applications, you need the consistency across languages that LE/VSE provides.

LE/VSE can respond in many ways to a condition. For example, LE/VSE can invoke a *condition handler*, a term used to define the specific routine that actually recognizes and responds to the condition. A condition handler can be registered by the CEEHDLR (register user-written condition handler) service, or be part of the language-specific condition handling services, such as a C signal handler or a PL/I ON-unit. HLL condition handling semantics that are intrinsic to the programming language also exist; an example is the COBOL ON SIZE phrase.

Related Run-Time Options and Callable Services

Here is a complete list of the run-time options and callable services that will help your application program detect and handle conditions. The new callable services available with these enhancements are CEE5GRO, CEE5SRP, and CEEMRCE.

Run-Time Options

ABPERC
Exempts from LE/VSE condition handling a single abend or program check.

DEPTHCONDLMT
Indicates how deep conditions might be nested.

ERRCOUNT
Indicates how many severity 2, 3, and 4 conditions can occur before issuing an abend.

TRAP
Indicates whether LE/VSE routines should handle abends and program interrupts: TRAP(ON,MIN) must be in effect for applications to run successfully.

USRHDLR
Registers a user condition handler at stack frame 0, allowing you to register a user condition handler without having to include a call to CEEHDLR in your application, and then recompile the application.

XUFLOW
Indicates whether exponent underflow should cause a program interrupt

Callable Services

CEE5CIB
Returns a pointer to the condition information block that is associated with a condition token passed to a user-written condition handler.

CEE5GRN
Gets the name of routine that incurred the condition currently being processed.

CEE5GRO
Gets the offset within the routine of the instruction that incurred the condition currently being processed.

CEE5SPM
Queries or modifies (by enabling or masking) hardware conditions.

CEE5SRP
Sets a position within a routine for execution to resume after a condition has been handled.

CEEDCOD
Decomposes or alters an existing condition token.

CEEGPID
Retrieves the version ID and the platform ID of the version and platform of LE/VSE currently in use.

CEEGQDT Retrieves the q_data token from the ISI

CEEHDLR Registers a user-written condition handler.

CEEHDLU Unregisters a user-written condition handler.

CEEITOK Returns the initial condition token from the current condition information block.

CEEMRCE Moves the resume cursor to an explicit location in a specific routine, where the location was previously defined by the CEE5SRP service.

CEEMRCR Moves the resume cursor relative to handle cursor (you might view this as performing a `GOTO` out of block in PL/I, or `setjmp()` and `longjmp()` in C).

CEENCOD Dynamically constructs a condition token.

CEESGL Signals a condition.

See *LE/VSE Programming Reference* for more information on run-time options and callable services.

The Stack Frame Model

A stack consists of an ordered set of stack elements, called stack frames, which are managed in a last-in first-out manner. In this book, unqualified references to *stack* mean *invocation stack*. The invocation stack can contain multiple *invocation stack frames*, which represent invocation instances of routines. A stack frame is added to the stack on entry to a routine and removed from the stack on exit from the routine.

The LE/VSE condition handling model is based on stack frames, in which condition handling can be different in different stack frames. Another condition handling model is global condition handling, which means that one condition handling mechanism remains in effect for the life of an application. The distinction between global condition handling and condition handling within a stack frame-based model can affect how a condition is handled in your application, particularly if it is a mixed-language application.

The following cause a stack frame to be added to the invocation stack:
- A function call in C that has not been inlined
- Entry into a program in COBOL
- Entry into a procedure or begin block in PL/I
- Entry into an ON-unit in PL/I

A stack frame is added to the stack every time a new routine is entered and removed when it is exited. LE/VSE uses stack frames to keep track of such things as the routine currently executing, the point at which an error occurs, and the point at which execution should resume after the condition is handled.

Each new stack frame may contain user-written condition handlers registered with CEEHDLR, but language-specific handlers such as C signal handlers are not associated with each stack frame. User condition handlers can be unregistered explicitly (by calling CEEHDLU) or implicitly, as when the routine that registered the handler returns control to its caller.

Two cursors, or pointers, keep track of the state of condition handling. The cursors are named the *handle* and *resume* cursors.

The Handle Cursor

If a condition occurs or is raised, the handle cursor initially points to the most recently established condition handler within the stack frame. As condition handling progresses, the handle cursor moves to earlier handlers within the stack frame, or to the first handler in the calling stack frame.

The Resume Cursor

The resume cursor points to the next sequential instruction where a routine would continue running if it were to resume.

Initially, the resume cursor is positioned after the machine instruction that caused or signaled the condition. You can move the resume cursor relative to the handle cursor by calling CEEMRCR. You can also use CEEMRCE to move the resume cursor to an explicit location in the application. You must have previously defined this location using a call to CEE5SRP (set resume point).

What Is a Condition in LE/VSE?

LE/VSE defines a *condition* as any event that can require the attention of a running application or the HLL routine supporting the application. A condition is also known as an exception, interrupt, or signal. LE/VSE makes it possible to respond to events that in the past might have caused a routine to abend, including hardware-detected errors and operating system-detected errors.

All of the following can generate a condition in LE/VSE:

Hardware-detected errors
Also known as program interruptions, these are signaled by the central processing unit. Examples are the fixed-overflow and addressing exceptions. The operating system derives the error codes from the codes defined for the machine on which the application is running. The error codes differ from machine to machine.

Operating system-detected errors
These are software errors and are reported to you as abends. An example is an OPEN error.

Software-generated signals
Signals are conditions intentionally and explicitly created by LE/VSE (using CEESGL), language library routines, language constructs (such as C's `raise()` or PL/I's SIGNAL), or user-written condition handling routines.

Under LE/VSE, an *exception* is the original event, such as a hardware signal, software-detected event, or user-signaled event, that is a potential condition. Through the enablement step (described briefly in "Steps in Condition Handling" on page 107 and in detail in the next chapter), LE/VSE might deem an exception to be a condition, at which point it can be handled by LE/VSE, user-written condition handlers, if they are present, or HLL condition handling semantics.

Steps in Condition Handling

LE/VSE condition handling is performed in three distinct steps: the enablement, condition, and termination imminent steps.

During the condition and termination imminent steps, the stack is used to determine the order of condition handler processing. Condition handlers associated with the most recent stack frame added to the stack are given first chance to handle the condition. Condition handlers associated with the next stack frame are next given a chance, and so on until either the condition is handled or default LE/VSE condition handling semantics take effect.

Only routines that are currently active on the stack have an effect on condition handling. For example, in a COBOL—PL/I application, a COBOL main program calls a PL/I subroutine. The subroutine then returns control to COBOL. The PL/I routine is no longer on the stack and does not affect condition handling.

Enablement Step

Enablement refers to the determination that an exception should be processed as a condition. The enablement step begins at the time an exception occurs in your application. In general, you are not involved with the enablement step; LE/VSE determines which exceptions should be enabled (treated as conditions) and which should be ignored, based on the languages currently active on the stack. If you do not specify explicitly or as a default any of the services or constructs discussed below, the default enablement of your HLL applies.

If LE/VSE ignores an exception, the exception is not seen as a condition, and will not undergo condition handling. Processing resumes at the next sequential instruction.

You can affect the enablement of exceptions in the following ways:

- Set the TRAP run-time option to handle or ignore abends and program checks.

  See "TRAP Effects on the Condition Handling Process" on page 108 for more information.
- Specify in the assembler user exit or ABPERC run-time option a VSE cancel code, a program-interruption code, or a user abend code to be exempted from LE/VSE condition handling.

  See "LE/VSE Abends and the Enablement Step" on page 108 for more information.
- Disable specific conditions by doing one of the following:
  - Code a construct such as `signal(sigfpe, SIG_IGN)` in a C function or a PL/I NOZERODIVIDE prefix in a PL/I procedure to request that program checks (in this case divide-by-zero) be ignored if they occur in either routine. Execution continues at the next sequential instruction after the one that caused the divide-by-zero. Condition handlers never get a chance to handle the program check because it is not considered a condition.
  - Call the CEE5SPM callable service or use the XUFLOW run-time option to disable hardware conditions.

    See "Using XUFLOW and CEE5SPM to Enable and Disable Hardware Conditions" on page 108 for more information.

In summary, not all hardware interrupts, software conditions, or user-signaled events become conditions. Those that are not ignored and do become conditions enter the condition step. See "Condition Step" on page 109 for the details of what takes place during the condition step.

TRAP Effects on the Condition Handling Process

The TRAP run-time option specifies how LE/VSE handles abends and program interrupts. In LE/VSE, TRAP(ON,MIN) must be in effect for applications to run successfully; TRAP(ON,MAX) is the IBM-supplied default.

Use the ABPERC runtime option to bypass LE/VSE condition handling for specific conditions, and to allow normal z/VSE processing to occur. Alternatively, use TRAP(ON,MIN) to generate a standard z/VSE system dump, for application failures. Do not use TRAP(OFF) unless requested to by IBM support personnel.

When TRAP(ON) is in effect, LE/VSE is notified of abends and program interrupts. Language semantics, C signal handlers, PL/I ON-units, and user-written condition handlers can then be invoked to handle them.

CEESGL and TRAP: When a condition is raised using the CEESGL callable service, C signal handlers, PL/I ON-units, and user-written condition handlers are always invoked if present, regardless of the setting of TRAP. If none of these handle the condition, then HLL semantics' default action could be taken. See *LE/VSE Programming Reference* for more information about TRAP and CEESGL.

LE/VSE Abends and the Enablement Step

You can prevent LE/VSE from automatically handling certain exceptions by requesting that a VSE cancel code or codes, a program-interruption code or codes, or a user abend code or codes be exempted from LE/VSE condition handling. If an abend is exempted, neither LE/VSE nor an HLL can handle it; LE/VSE issues an operating system request to terminate the enclave.

Abends can be exempted from LE/VSE condition handling in two ways:

- You can specify in the assembler user exit CEEBXITA a list of VSE cancel codes, program-interruption codes, user abend codes, or a combination of any of these codes, that LE/VSE will exempt from normal condition handling. See Chapter 25, "Using Run-Time User Exits," on page 319 for more information.
- The ABPERC run-time option allows you to specify which (if any) VSE cancel code, program-interruption code, or user abend code should be exempt from LE/VSE condition handling. ABPERC is intended for use as a debugging tool that allows the application to execute with TRAP(ON).

  For a list of LE/VSE-issued abends and information about using ABPERC to debug your application, see *LE/VSE Debugging Guide and Run-Time Messages*. For a list of VSE cancel codes, see *z/VSE Messages and Codes* For a list of program-interruption codes, see the *Principles of Operation* manual for your machine. For more information about the TRAP and ABPERC run-time options, see *LE/VSE Programming Reference*.

Using XUFLOW and CEE5SPM to Enable and Disable Hardware Conditions

You can change the enablement of certain hardware interrupts using the CEE5SPM callable service and XUFLOW run-time option (see *LE/VSE Programming Reference*).

LE/VSE provides the CEE5SPM callable service to replace assembler language routines that manipulate bits 20 through 23 of the Program Status Word (PSW) to enable or disable the following hardware interrupts:

Decimal overflow
Exponent underflow
Fixed-point overflow
Significance

The XUFLOW run-time option specifies whether or not an exponent underflow exception causes a program interrupt. Both CEE5SPM and XUFLOW can change the condition handling semantics of the HLL or HLLs of your application. Therefore, use CEE5SPM and XUFLOW only if you understand the effect they have on your application.

PL/I Considerations: PL/I semantics depend on the program mask being given certain settings:

- The fixed-point overflow, decimal overflow, and exponent underflow masks are ON
- The significance mask is OFF

C Considerations: C ignores requests to enable the decimal overflow, exponent underflow, fixed-point overflow, or significance exceptions.

COBOL Considerations: The decimal overflow and fixed-point overflow exceptions **cannot** be enabled in a COBOL routine; COBOL ignores any request to enable these exceptions.

Condition Step

The condition step begins after the enablement step has completed and LE/VSE determines that an exception in your application should be handled as a condition. In the simplest form of this step, LE/VSE traverses the stack beginning with the stack frame for the routine in which the condition occurred and progresses towards earlier stack frames. Condition handlers are invoked at each intervening stack frame and given a chance to respond in any of the ways described in "Responses to Conditions" on page 116. The condition step lasts until a condition handler requests a resume or until default condition handling occurs (condition went unhandled).

Throughout the following discussion, it might help you to refer to Figure 31.

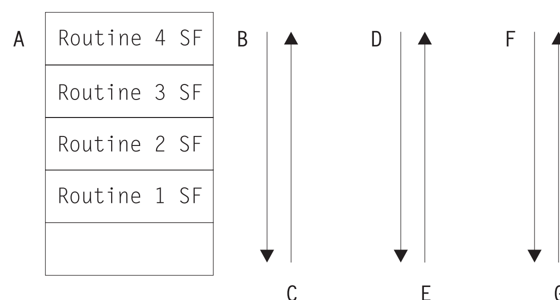


Figure 31. Condition Processing

1. LE/VSE condition handling begins at the most recently activated stack frame. This is the stack frame associated with the routine that incurred the condition. In Figure 31, this is **A**, or routine 4.

2. If a debug tool, such as Debug Tool for VSE/ESA, is present, and the setting of the TEST run-time option indicates that it should be given control, it is invoked. See *LE/VSE Programming Reference* for information about the TEST run-time option.
3. If the debug tool is not invoked, or does not handle the condition, LE/VSE traverses the stack, stack frame by stack frame, towards earlier stack frames. This is in the direction of arrow **B** in Figure 31 on page 109. First user-written condition handlers established using CEEHDLR, and then language-specific condition handlers present at each stack frame, such as C signal handlers or PL/I ON-units, can all respond by percolating, promoting, or handling the condition (see "Responses to Conditions" on page 116 for a discussion of these actions).
4. Condition handling is complete if one of the handlers requests the application to resume execution. If all stack frames have been visited, and no condition handler has requested a resume, the language of the routine in which the exception occurred can enforce default condition handling semantics.
5. If the HLL of the routine that originated the condition does not issue a resume, what occurs next depends on whether there is a PL/I routine active on the stack.
   a. The condition is percolated if there is no currently active PL/I routine or if the condition is not one that PL/I promotes to the ERROR condition (see "Promoting Conditions to the PL/I ERROR Condition" on page 133 for details). LE/VSE default actions are then taken based on the severity of the unhandled condition, as indicated in Table 28 on page 111. If the condition is of severity 2 or above, LE/VSE promotes the condition to T_I_U (termination imminent due to an unhandled condition) and returns to routine 4 to redrive the stack (this occurs at points **C** and **D** in Figure 31 on page 109). For more information about the termination imminent step and T_I_U, see "Termination Imminent Step" on page 111.
   b. If the condition is one that PL/I promotes to the PL/I ERROR condition (see "Promoting Conditions to the PL/I ERROR Condition" on page 133 for details), the condition is promoted at the location represented as **C** in Figure 31 on page 109, and another pass is made of the stack. The following takes place:

      On the next pass of the stack (**D**), any ERROR ON-unit or user-written condition handler is invoked. If the ON-unit or user-written condition handler issues a resume, condition handling ends. Execution resumes where the resume cursor points.

      If no ON-unit or user-written condition handler issues a resume, the ERROR condition is promoted (at **E**) to T_I_U. (See "Processing the T_I_U Condition" on page 112 for a discussion of T_I_U.)

      A final pass of the stack is made, beginning in Routine 4 where the original condition occurred (**F**). Because T_I_U maps to the PL/I FINISH condition, both established PL/I FINISH ON-units and user-written condition handlers registered for T_I_U are invoked.

      If no user-written or HLL condition handlers act on the condition, LE/VSE begins thread termination activities in response to the unhandled condition (**G**). See Table 28 on page 111 for the default actions that LE/VSE takes for conditions of different severities.

Influencing Condition Handling with the ERRCOUNT Run-Time Option

The ERRCOUNT option allows you to specify the number of errors that are tolerated during the execution of an enclave. Each condition of severity 2 or above,

regardless of its origin, increments the error count by one. If the error count exceeds the limit, LE/VSE terminates the enclave with abend code 4091 and reason code 11.

See *LE/VSE Programming Reference* for syntax and more information on using ERRCOUNT.

Table 28. LE/VSE Default Responses to Unhandled Conditions. LE/VSE's default responses to unhandled conditions fall into one of two types, depending on whether the condition was signaled using CEESGL and an fc parameter, or the condition came from any other source.

Severity of Condition	Condition Signaled by User in a Call to CEESGL with an fc	Condition Came from Any Other Source
0 (Informative message)	Return CEE069 condition token, and resume processing at the next sequential instruction. See the fc table for CEESGL (*LE/VSE Programming Reference*) for a description of the CEE069 condition token.	Resume without issuing message.
1 (Warning Message)	Return CEE069 condition token, and resume processing at the next sequential instruction.	If the condition occurred in a stack frame associated with a COBOL program, resume and issue the message. If the condition occurred in a stack frame associated with a non-COBOL routine, resume without issuing message.
2 (Program terminated in error)	Return CEE069 condition token, and resume processing at the next sequential instruction.	Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified. (See "Processing the T_I_U Condition" on page 112 for more information on T_I_U.)
3 (Program terminated in severe error)	Return CEE069 condition token, and resume processing at the next sequential instruction.	Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.
4 (Program terminated in critical error)	Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.	Promote condition to T_I_U, redrive the stack, then terminate the thread if the condition remains unhandled. Message issued if TERMTHDACT(MSG) is specified.

Termination Imminent Step

The termination imminent step occurs for certain unhandled conditions or as the result of STOP-like language constructs such as PL/I STOP, C `exit()` or `abort()`, or COBOL STOP RUN. The termination imminent step occurs when one of the following events occurs:

- The T_I_U condition (Termination Imminent due to Unhandled condition) is raised
- The T_I_S condition (Termination Imminent due to Stop) is raised to indicate that the thread can potentially terminate

When T_I_U or T_I_S is raised, another pass is made of the stack. See "Processing the T_I_U Condition" on page 112 and "Processing the T_I_S Condition" on page 112 for details on what can happen during and after the pass.

You can directly signal T_I_U and T_I_S using the CEESGL callable service. When you do, LE/VSE behaves in the way described in "CEESGL and the Termination Imminent Step" on page 114.

Processing the T_I_U Condition

Table 28 on page 111 indicates that for severity 4 conditions signaled by CEESGL, and for severity 2 and above conditions that remain unhandled after all condition handlers have had a chance to handle them, LE/VSE promotes the unhandled condition to T_I_U. See *LE/VSE Programming Reference* for a discussion of CEESGL.

T_I_U is a severity 3 condition with the representation shown in Table 29:

Table 29. T_I_U Condition Representation

Symbolic Feedback Code (fc)	Severity	Message Number	Message Text
CEE066	3	0198	Termination of a thread was signaled.

After promoting the condition to T_I_U, LE/VSE does the following:

1. LE/VSE revisits each stack frame on the stack, beginning with the stack frame in which the condition occurred, and progressing towards earlier stack frames. At each stack frame, HLL and user-written condition handlers are given a chance to handle the condition.

   The T_I_U condition maps to the PL/I FINISH condition. Therefore, an established PL/I FINISH ON-unit or registered user-written condition handler can be invoked to handle the condition. After the ON-unit or condition handler completes its processing, the termination activities described in step 3 take place.
2. If, during the course of condition handling, the resume cursor is moved and a resume is requested by a condition handler, execution resumes at the instruction pointed to by the resume cursor. If a resume is requested for the T_I_U condition without moving the resume cursor, the thread terminates immediately with no clean-up. See *LE/VSE Programming Reference* for a discussion of the CEEMRCR service.
3. If all stack frames have been visited, and the condition remains unhandled, or a FINISH ON-unit or user-written condition handler has processed the condition and returned, LE/VSE performs the following termination activities:
   - Sets the reason and return codes. The return code value is based on the severity of the original unhandled condition, not on the T_I_U condition (which is a severity 3).
   - Issues a message for the condition.
   - Prints a traceback and dump depending on the setting of the TERMTHDACT run-time option (see *LE/VSE Programming Reference* for syntax).
   - Terminates the thread. (This release of LE/VSE only supports a single thread within an enclave. Therefore, when a thread terminates, the entire enclave terminates.)

Processing the T_I_S Condition

The termination imminent step of condition handling can also be entered as the result of the T_I_S (Termination_Imminent due to STOP) condition being signaled. T_I_S is a severity 1 condition with the representation shown in Table 30 on page 113:

Table 30. T_I_S Condition Representation

Symbolic Feedback Code (fc)	Severity	Message Number	Message Text
CEE067	1	0199	Termination of a thread was signaled.

The T_I_S condition is raised by LE/VSE immediately upon detection of a language STOP-like construct such as:

C `exit()` function
COBOL STOP RUN
PL/I EXIT statement
PL/I STOP statement

The HLL constructs listed above initiate termination activities for the enclave in two steps:

1. LE/VSE traverses the stack beginning at the stack frame for the routine containing the STOP-like statement and proceeds, stack frame by stack frame, towards earlier stack frames. User-written and HLL condition handlers at each stack frame are given a chance to handle the condition.

   T_I_S maps to the PL/I FINISH condition. Therefore, both established PL/I FINISH ON-units and user-written condition handlers can be invoked. After the ON-unit or condition handler completes its processing, the termination activities described in step 2 take place.
2. If all stack frames have been visited, and the condition remains unhandled, or an ON-unit or condition handler has processed the condition and returned, LE/VSE:
   - Sets the reason and return codes
   - Terminates the thread

   LE/VSE performs only one pass of the stack for STOP-like statements.

The Termination Imminent Step and the TERMTHDACT Run-Time Option

You can use the TERMTHDACT run-time option to set the type of information you receive after your application terminates in response to a severity 2, 3, or 4 condition. For example, you can specify that a message is issued or a dump is generated if the application terminates.

PL/I Considerations: For those PL/I conditions that do not raise the ERROR condition as part of their implicit action, PL/I requires that a message be issued. For these conditions, the message is issued regardless of the setting of TERMTHDACT. Therefore, messages may be delivered even when TERMTHDACT(QUIET) is set.

If the condition remains unhandled (for example, the PL/I FINISH condition is still regarded as unhandled after normal return from a FINISH ON-unit), and the application terminates, the message associated with the condition is not issued again at termination.

For more information about TERMTHDACT, see *LE/VSE Debugging Guide and Run-Time Messages* and *LE/VSE Programming Reference*.

CEESGL and the Termination Imminent Step

You can signal T_I_U and T_I_S directly with the CEESGL callable service. Two reasons you might need to do this are:

- To force the driving of a FINISH ON-unit or similar construct that would perform clean-up activities
- To test a PL/I ON-unit or user-written condition handler that you have designed to handle T_I_U or T_I_S

If you signal T_I_U or T_I_S by calling CEESGL with the feedback code parameter, the following occurs:

1. LE/VSE visits each stack frame on the stack, beginning with the stack frame in which the condition was signaled, and progressing towards older stack frames. At each stack frame, HLL and user-written condition handlers are given a chance to handle the condition.

   T_I_U and T_I_S both map to the PL/I FINISH condition. Therefore, an established PL/I FINISH ON-unit can be invoked to handle the condition.
2. If all stack frames have been visited, and the condition remains unhandled, or a FINISH ON-unit has processed the condition and returned, LE/VSE returns the CEE069 condition token to the routine that called CEESGL, and processing resumes at the next sequential instruction.

Invoking Condition Handlers

After a condition has been enabled, LE/VSE steps through the stack and passes control to the most recently established condition handling routines in the stack. Condition handling routines can be in the form of a debug tool, a user-written condition handler, or a language-specific condition handling mechanism:

Debug Tool
If you have invoked a debug tool using the TEST run-time option or the CEETEST callable service (see *LE/VSE Programming Reference*), the debug tool gains control when a condition occurs. Unless a condition is promoted and is passed through the stack again for additional condition handling, a debug tool is invoked only once per stack.

User-Written Condition Handler
User-written condition handlers are routines that you supply to handle specific conditions that might arise in the run-time environment. As shown in Figure 32 on page 115, a LIFO queue containing zero or more user-written condition handlers is associated with each stack frame. A different queue exists for each stack frame. For example, if routine A calls routine B, there is a new queue associated with the stack frame for routine B.

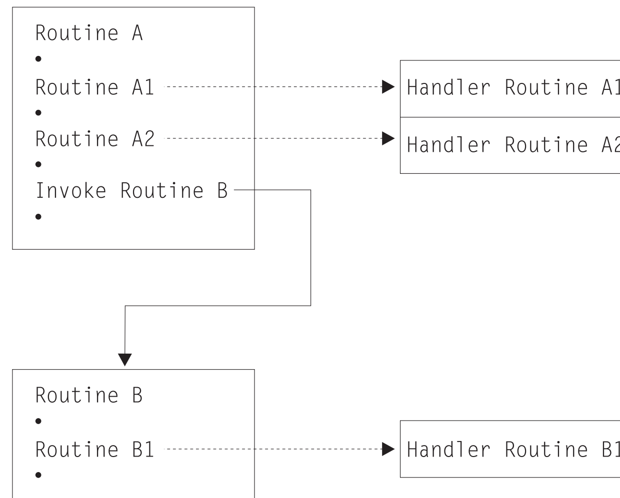


Figure 32. Queues of User-Written Condition Handlers

User-written condition handlers are registered on a stack frame-by-stack frame basis using the CEEHDLR callable service. A call to CEEHDLR from a given routine adds a user-written condition handler onto the queue for the stack frame associated with that routine. Registering a condition handling routine using CEEHDLR implicitly requests LE/VSE to pass control to this routine when a condition occurs. For example, you could call CEEHDLR to register two user-written condition handlers for the same stack frame, one that handles floating point underflow conditions and another that handles floating point divide conditions.

The most recent user condition handler registered using CEEHDLR is the first to be invoked by LE/VSE. Note that you could also register a single user condition handler to handle both of these conditions.

The user-written condition handlers can respond to a condition in any of the ways described in the section “Responses to Conditions” on page 116.

User-written condition handlers are given a chance to handle a given condition before the language-specific condition handling semantics described below take effect.

Language-Specific Condition Handling Semantics

If language-specific semantics are established within a stack frame, they are honored. Of course, the language-specific handling mechanisms act only on those conditions for which the language has a defined action. The language *percolates* all other conditions by simply passing them on to the next condition handler.

If a condition is unhandled after the stack is traversed, default language-specific and LE/VSE condition semantics take over.

Responses to Conditions

Condition handlers are routines written to respond to conditions in one of the following ways:

Resume
A resume occurs when a condition handler determines that the condition was handled and normal application execution should resume. A program resumes running usually at the instruction immediately following the point where the condition occurred.

A resume cursor points to the place where a routine should resume. The resume cursor can be manipulated to be placed at a specific point by using the CEEMRCR callable service (see *LE/VSE Programming Reference*).

Percolate
A condition is percolated if a condition handler declines to handle it. User-written condition handlers, for example, can be written to act on a particular condition, but percolate all other conditions. LE/VSE can continue condition handling in one of the following places:

- With the next condition handler associated with the current stack frame. This can be either the first condition handler in a queue of user-established condition handlers, or the language-specific condition semantics.
- With the most recently established condition handler associated with the calling stack frame.

Promote
A condition is promoted when a condition handler converts the condition into one with a different meaning. A condition handler can promote a condition for a variety of reasons, including the condition handler's knowledge or lack of knowledge about the cause of the original condition. A condition can be promoted to simulate conditions that would normally come from a different source.

Condition Handling Scenarios

The following condition handling scenarios can help you better understand what occurs during the condition handling steps. The scenarios differ in complexity, with Scenario 1 being the easiest to understand. The scenarios assume that no debug tool has been invoked.

See Chapter 12, "LE/VSE and HLL Condition Handling Interactions," on page 121 if you are interested in specific HLL condition handling behavior.

Scenario 1: Simple Condition Handling

Refer to Figure 33 on page 117 throughout the following discussion.

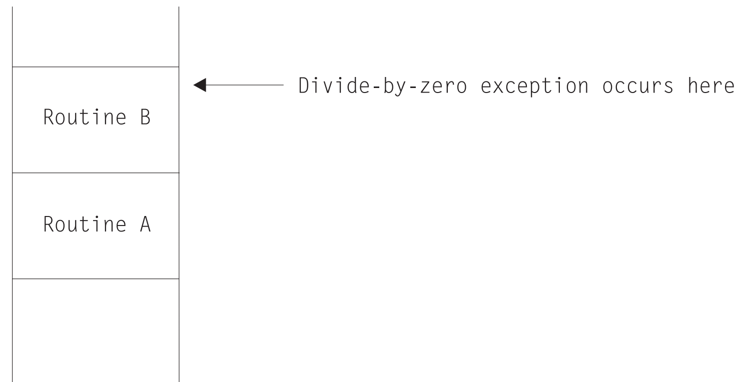


Figure 33. Scenario 1: Division by Zero with No User Condition Handlers Present

In this scenario, no C handlers created by a call to `signal()`, PL/I ON-units, or user-written condition handlers registered using the CEEHDLR service are established at any stack frame in the application.

1. A divide-by-zero exception occurs in routine B.
2. The divide-by-zero exception is enabled by the language of the stack frame in which it occurred because it is a problem that, if it remains unhandled, causes termination.
3. The following occurs in the condition step:
   - If any user-written condition handlers have been registered using the CEEHDLR callable service on routine B's stack frame, they are given control. No handlers have been registered, so the condition is percolated.
   - If a C signal handler is registered, or if a PL/I ON-unit is established on the stack frame, it is given control. Neither one exists on routine B's stack frame, so the condition is percolated.
   - If any user-written condition handlers have been registered using CEEHDLR on routine A's stack frame, they are given control. None have been, so the condition is percolated.
   - If a PL/I ON-unit is established on routine A's stack frame, it is given control. No PL/I ON-unit has been established for the stack frame, so the condition is percolated.
   - After the oldest stack frame (in this case, that for routine A) has been checked, HLL and LE/VSE default actions occur. Assume that the HLL percolates the condition to LE/VSE.

     LE/VSE examines the severity of the unhandled divide-by-zero condition (severity 3), promotes the condition to T_I_U, and requests that the stack be redriven. This is the end of the condition step and the beginning of the termination imminent step.
4. The following occurs during the termination imminent step:
   - The stack frame for routine B is revisited, and if a user-written condition handler is present, it is given control. No handlers are registered, so T_I_U is percolated.
   - If a C signal handler or PL/I ON-unit can respond to the T_I_U condition, it is given control. In this case, there isn't one, so the condition is percolated.
   - The stack frame for routine A is revisited, and checked for user-written condition handlers registered for the T_I_U condition, C signal handlers or PL/I ON-units. No handlers are registered, so T_I_U is percolated.

- LE/VSE takes the default action for the unhandled T_I_U condition, which terminates the enclave.

Scenario 2: Condition Handling with User-Written Condition Handler Present for T_I_U

Scenario 2 is much the same as scenario 1, except that routine A does have a user-written condition handler established. Refer to Figure 34 throughout the following scenario.

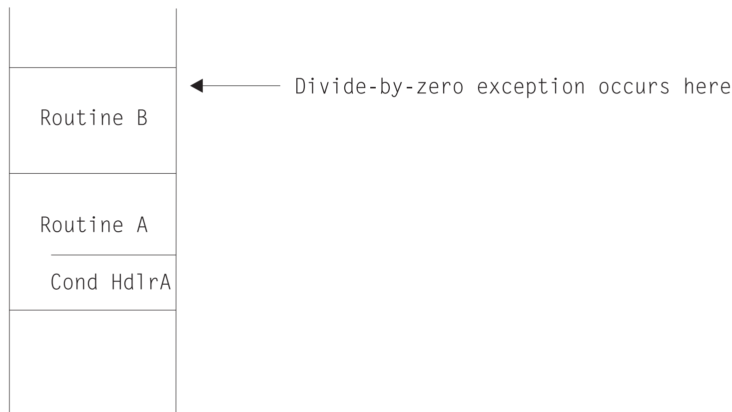


Figure 34. Scenario 2: Division by Zero with a User-Written Condition Handler Present in Routine A

In this scenario, routine A is a routine that invokes other prewritten applications. If any of the components of the prewritten application fail, routine A must remain up and take alternative action. Therefore, routine A has a user-written condition handler registered. The handler is designed to handle the T_I_U condition by issuing a nonlocal jump to a location within routine A. The handler percolates all conditions other than T_I_U.

1. A divide-by-zero exception occurs in routine B.
2. The divide-by-zero exception is enabled by the language of the stack frame in which it occurred because it is a problem that, if it remains unhandled, causes termination.
3. The following occurs in the condition step:
   - If any user-written condition handlers have been registered using the CEEHDLR callable service on routine B’s stack frame, they are given control. No handlers have been registered, so the condition is percolated.
   - If a C signal handler has been registered or a PL/I ON-unit has been established for the divide-by-zero condition, it is given control. No C signal handler or PL/I ON-unit is present, so the condition is percolated to LE/VSE.
   - A user-written condition handler has been registered on routine A’s stack frame, so it is given control. However, because the divide-by-zero condition is not the one the handler is looking for, the condition is percolated.
   - If a C signal handler is registered or a PL/I ON-unit is established for the condition on routine A’s stack frame, it is given control. Neither one is present, so the condition is percolated.

- After the earliest stack frame (in this case, that for routine A) has been checked, HLL and LE/VSE default actions occur. In this case, assume that the HLL percolates the condition to LE/VSE.

  LE/VSE examines the severity of the unhandled divide-by-zero condition (severity 3), promotes the condition to T_I_U, and requests that the stack be redriven. This is the end of the condition step and the beginning of the termination imminent step.

4. The following occurs during the termination imminent step:
   - LE/VSE revisits the stack frame for routine B, checking for user-written condition handlers registered for the T_I_U condition. No handlers are registered, so T_I_U is percolated.
   - If a PL/I FINISH ON-unit is present, it is given control. In this example, there isn't one, so the condition is percolated.
   - LE/VSE revisits the stack frames for routine A, checking for user-written condition handlers registered for the T_I_U condition. There is one, and it is given control. The user code in the handler, using either HLL or LE/VSE facilities, causes control to pass to a location within routine A.
5. Control resumes with routine A at the location specified. The condition is now handled.

Scenario 3: Condition Handling with User-Written Condition Handler Present for Divide-by-Zero Condition

Scenario 3 is much the same as scenario 2, except that routine B has a user-written condition handler established to handle the divide-by-zero condition.

Refer to Figure 35 throughout the following scenario.

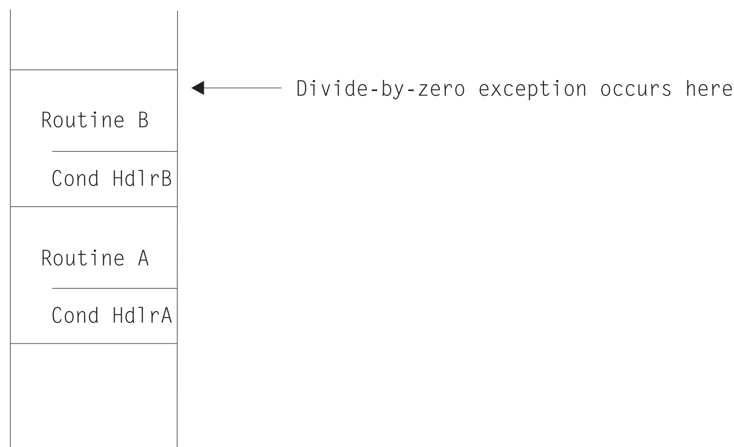


Figure 35. Scenario 3: Division by Zero with a User Handler Present in Routine B

The handler established by routine B is designed to deal with divide-by-zero and possibly other conditions that occur either during its execution or in the routines that it calls. For a divide-by-zero condition, the handler is to print a message and continue processing.

1. A divide-by-zero exception occurs in routine B.
2. The divide-by-zero exception is enabled by the language of the stack frame in which it occurred because it is a problem that, if it remains unhandled, causes termination.

3. The following occurs in the condition step:
   - A user-written condition handler has been registered using the CEEHDLR callable service on routine B's stack frame, so it is given control. The handler recognizes the divide-by-zero as a condition it is capable of dealing with. It produces a message, does appropriate clean-up, and then causes resumption either through HLL constructs or LE/VSE services.
4. The condition is now considered to be handled and is never seen by stack frame A or the LE/VSE default handler.

Chapter 12. LE/VSE and HLL Condition Handling Interactions

This is the second part of the condition handling discussion. It would be helpful for you to read Chapter 11, "LE/VSE Condition Handling Introduction," on page 103 before reading this chapter. Chapter 11 introduces you to terminology and concepts that are discussed in the present chapter, and offers a brief overview of pre-LE/VSE HLL condition handling. It discusses in detail the LE/VSE condition handling model and the many services that you can use to tailor how conditions are handled in your application. In addition, it introduces the three steps of condition handling in LE/VSE.

Understanding the Basics

This chapter discusses HLL condition handling semantics, focusing on how HLL semantics interact with the LE/VSE condition handling model and services. C, COBOL, and PL/I are each discussed, and condition handling scenarios and examples are provided. See one of the following sections for details:

- "C Condition Handling Semantics"
- "COBOL Condition Handling Semantics" on page 128
- "PL/I Condition Handling Semantics" on page 132

If you are running a single-language application written in a language such as C or PL/I that has extensive built-in error handling functions, and you are relying entirely upon the semantics of these languages to handle errors, you will not notice much difference in how errors are handled under LE/VSE.

On the other hand, if you are running a single-language application written in a language such as COBOL or assembler that has little built-in error handling, you might notice a change in how errors are handled under LE/VSE. For example, in an application that relies on abend codes to handle errors, you might need to alter the assembler user exit to get the same behavior under LE/VSE as under the previous run-time environment. See Chapter 25, "Using Run-Time User Exits," on page 319 for information on modifying the assembler user exit.

For information on condition handling in ILC applications, see *LE/VSE Writing Interlanguage Communication Applications*.

C Condition Handling Semantics

This section describes C condition handling.

C employs a global condition handling model, which, on initialization, defines the actions that are taken when a condition is raised. The actions defined by C apply to an entire enclave, not just to a routine or block within an enclave. You can alter a specific action that the C condition handler takes when a condition is raised, however, by coding `signal()` function calls in your applications.

C recognizes a number of errors; some correspond directly to the errors detected by the hardware or the operating system, and some are unique to C. All actions for condition handling are controlled by the contents of the C global error table. Table 31 on page 122 contains default C-language error handling semantics.

© Copyright IBM Corp. 1991, 2005

Table 31. C Conditions and Default System Actions

C Condition	Origin	Default Action
SIGILL	Execute exception Operation exception Privileged operation `raise(SIGILL)`	Abnormal termination (return code=3000)
SIGSEGV	Addressing exception Protection exception Specification exception `raise(SIGSEGV)`	Abnormal termination (return code=3000)
SIGFPE	Data exception Decimal divide Exponent overflow Fixed-point divide Floating-point divide `raise(SIGFPE)`	Abnormal termination (return code=3000)
SIGABRT	`abort()` function `raise(SIGABRT)`	Abnormal termination (return code=2000)
SIGABND	Abend the function	Abnormal termination (return code=3000)
SIGTERM	Termination request `raise(SIGTERM)`	Abnormal termination (return code = 3000)
SIGINT	Attention condition	Abnormal termination (return code = 3000)
SIGIOERR	I/O errors	Ignore the condition
SIGUSR1	User-defined condition	Abnormal termination (return code=3000)
SIGUSR2	User-defined condition	Abnormal termination (return code=3000)
Masked	Exponent overflow Fixed-point underflow Significance	These exceptions are disabled. They are ignored during the condition handling process, even if you try to enable them using the CEE5SPM callable service.

Comparison of C-LE/VSE Terminology

The term *signal* is defined differently under C than under LE/VSE, and you need to know the distinction to understand how C and LE/VSE condition handling interact. Here is a comparison of the terminology LE/VSE and C use to describe the same general idea:

- Using LE/VSE services, you *register* a condition handler by using CEEHDLR, and you *raise* a condition by using CEESGL.
- Using C functions, you *register* a signal handler by using the `signal()` function, and you *raise* a signal using the `raise()` function.

  You can think of *signal* as the C term for an LE/VSE *condition*. To simplify the following discussion, the term *condition* is used in place of *signal*.

Controlling Condition Handling in C

In C, conditions can come from two main sources:

- An exception might occur because of an error in the code. The exception might or might not be seen as a condition, depending on how you use the `signal()` function.

- You can explicitly report a condition by using the raise() function.

Using the signal() Function

The C signal() function call alters the actions that the global error table specifies will be taken for a given condition. You can use signal() to do the following:

- Ignore the condition completely. You do this by specifying signal(sig_num,SIG_IGN), where sig_num represents the condition to be ignored. When the action for the condition is to ignore it, the condition is considered to be *disabled*. The condition will therefore not be seen.

  Note: Exceptions to this rule are:
  - The SIGABND condition
  - A system or user abend represented by LE/VSE message CEE3250C under CICS
  - A system abend (other than a program check) represented by LE/VSE message CEE3321C in batch
  - A user abend represented by LE/VSE message CEE3322C in batch

  These are never ignored, even if you specify SIG_IGN in a call to signal().
- Reset condition handling to the defaults shown in Table 31 on page 122. Actions for handling a condition are implicitly reset to the system default when the condition is reported, but at times you need to explicitly reset condition handling. Specify signal(sig_num, SIG_DFL), where sig_num represents the condition to be reset.
- Register a signal handler to handle the condition. Specify signal(sig_num, sig_handler), where sig_num represents the condition to be handled, and sig_handler represents a pointer to the user-written function that is called when the condition occurs.

  The signal handler specified in signal() is given a chance to handle a condition only after any user-written handler established using CEEHDLR is invoked.

Using the raise() Function

When the C raise() function is called for any of the conditions listed in Table 31 on page 122, a corresponding LE/VSE condition is automatically raised by a call to the CEESGL callable service. Any of these conditions (EDC6000 through EDC6004) can be handled by a user-written condition handler registered using the CEEHDLR service. For detailed descriptions of conditions EDC6000 through EDC6004, see *LE/VSE C Run-Time Programming Guide* and *LE/VSE Debugging Guide and Run-Time Messages*.

For more information about the CEEHDLR and CEESGL callable services, see *LE/VSE Programming Reference*. For more information about using the raise() function, see *LE/VSE C Run-Time Library Reference*.

C atexit() Considerations

In all C applications, the atexit list is honored only after all condition handling activity has taken place and all user code is removed from the stack, which invalidates any jump buffer previously established.

With C, you can register a number of routines that gain control during the termination of an enclave. When using the C atexit() function, consider the following:

- A C atexit routine can nominate only C routines, but those routines can call routines written in other languages.

- User-written condition handlers can be registered while running an `atexit` routine. However, any jump buffers established are invalid.
- If a severity 2 or greater condition arises while running an `atexit` routine and it is unhandled, further `atexit` routines are skipped and the LE/VSE environment is terminated.
- A C `exit()` function or PL/I STOP or EXIT statement issued within an `atexit` routine halts all other `atexit` functions.
- If, while running an `atexit` routine, an attempt to register another `atexit` routine is made, the registration is ignored. The `atexit` routine returns a nonzero result indicating a failure to register the routine.

C Condition Handling Actions

In this section the condition handling semantics of C-only applications are described as they relate to the LE/VSE condition handling model. Condition handling for applications with both C and non-C routines is discussed in *LE/VSE Writing Interlanguage Communication Applications*.

If an exception occurs while a C routine is executing, the following activities are performed:

1. The LE/VSE enablement step of condition handling is entered.

   If the action defined for the exception is to ignore it for one of the following reasons, the condition is disabled. Execution continues at the next sequential instruction after the point where the condition occurred.

   - You have specified `SIG_IGN` in a call to the `signal()` function for any C condition except the following:
     - The SIGABND condition
     - A system or user abend represented by LE/VSE message CEE3250C under CICS
     - A system abend (other than a program check) represented by LE/VSE message CEE3321C in batch
     - A user abend represented by LE/VSE message CEE3322C in batch
   - The exception is one of those listed as masked in Table 31 on page 122.
   - You did not specify any action, but the default action for the condition is `SIG_IGN` (see Table 31 on page 122).
   - You are running under CICS and a CICS handler is pending.
2. If SIG_IGN is not specified or defaulted for the exception, and the exception is not masked, the LE/VSE condition step of condition handling is entered. These activities then occur:
   - If a debug tool is present, and the setting of the TEST run-time option indicates that it should be given control, it is invoked. See *LE/VSE Programming Reference* for information about the TEST run-time option.
   - If a debug tool is not invoked, or does not handle the condition, any user-written condition handlers registered using CEEHDLR for that stack frame are invoked.
   - If no user-written condition handlers are registered for the condition that has occurred, and if you have registered a signal handler for the condition, that handler is invoked.
   - If the signal handler handles the condition, control returns to the routine in which the condition occurred. If the signal handler cannot handle the condition, it might force termination by issuing `exit()` or `abort()`, or might issue a `longjmp()`.

Condition handling can only continue after a signal handler gains control if you specify SIG_DFL in a call to signal(). If you do, the condition is percolated to the next user-written condition handler registered using CEEHDLR, or to the language-specific condition handler associated with the next stack frame.

- If condition handlers at every stack frame have had a chance to respond to the condition and it still remains unhandled, the LE/VSE default actions described in Table 28 on page 111 take place.
- If the LE/VSE default action is to promote the condition to T_I_U (Termination Imminent due to an Unhandled condition), the termination imminent step of condition handling is entered.

3. When the condition is promoted to T_I_U, LE/VSE makes another pass of the stack looking for user-written condition handlers registered for T_I_U.

   If, on the next pass of the stack, no condition handler issued a resume or moved the resume cursor, LE/VSE terminates the enclave.

C Condition Handling Examples

Condition Occurs with No Signal Handler Present: The following three examples illustrate how a condition such as a divide-by-zero is handled in a C routine in LE/VSE if you do not use any LE/VSE callable services, or don't have any user-written condition handlers registered.

There is no user-written condition handler or signal handler registered for C370C or any of the other C routines, so the condition is percolated through all of the stack frames on the stack. At this point, C default actions take place of percolating the condition to LE/VSE. LE/VSE takes its default action for an unhandled severity 3 condition and terminates the application. A message, dump, or trace could be generated depending on the setting of TERMTHDACT (see *LE/VSE Programming Reference*).

This example is a C main routine that calls C370B, a subroutine that passes data to another subroutine, C370C.

```
/*Module/File Name:  EDCMLTA  */
/*****************************************************************/
/* Demonstrate a failing C/370 program                           */
/* with multiple active routines                                 */
/* on the stack.  The call sequence is as follows:               */
/*  C370A ---> C370B ---> C370C (which does a divide-by-zero)    */
/*****************************************************************/

#include <stdio.h>

int y = 0;
void C370B(void);

int main(void) {

  printf("In Program C370A\n");
  C370B();
}
```

Figure 36. C370A Routine

This example is a C subroutine that calls C370C, and passes data to it.

```
/*Module/File Name:  EDCMLTB  */
/*********************************************************************/
/* This routine is called to pass data forward to C370C.             */
/* C370C will then cause a zero divide.                              */
/*********************************************************************/

#include <stdio.h>

extern int y;
void C370C(int);

void C370B(void) {

  int x;
  printf("In Program C370B\n");
  x = y;
  C370C(x);
}
```

Figure 37. C370B Routine

The following example generates a divide-by-zero. The divide-by-zero condition is percolated back to C370B, to C370A, and to LE/VSE default behavior.

```
/*Module/File Name:  EDCMLTC  */
/*********************************************************************/
/* This routine is called by C370B to generate a zero divide.        */
/*********************************************************************/

#include <stdio.h>


void C370C(int y) {

  printf("In Program C370C\n");
  y = 1/y;
}
```

Figure 38. C370C Routine

Condition Occurs with Signal Handler Present: The following contains a simple example of a C application in which `y = a/b` is a mathematical operation. `signal (SIGFPE, c_handler)` is a signal invocation that registers the routine `c_handler()` and gives it control if a floating-point divide exception occurs.

```
/*Module/File Name:  EDCCSIG  */
/*********************************************************************/
/* A routine with a C condition handler registered.                 */
/*********************************************************************/

#include <stdio.h>
#include <signal.h>

void c_handler(int);

int main(void) {
  int a=8, b=0, y;
  /* .
     .
     . */
  signal (SIGFPE, c_handler);
  /* .
     .
     . */
  y = a/b;
  /* .
     . */
}

void c_handler(int i)
{
   printf("handled SIGFPE\n");
  /* .
     . */
   return;
}
```

Figure 39. C Condition Handling Example

If b = 0, a floating-point divide condition occurs. LE/VSE condition handling begins:

- The enablement step occurs.
  - If Table 31 on page 122 indicates that floating-point divide is a masked exception, the exception is ignored. The floating-point divide is not a masked exception, however.
  - If `SIG_IGN` is specified for the `SIGFPE` exception in any of the three examples, then the `SIGFPE` exception is ignored. However, this does not occur.

  The floating-point divide condition is enabled and enters the condition step of condition handling.
- If a debug tool is present, it receives control.
- If a user-written condition handler is registered by CEEHDLR for that stack frame, it receives control.

If none of the above takes place, the condition manager gives the C signal-handler control. This handler in turn invokes `c_handler()` as specified in the `signal()` function in the Figure 39. Control is then returned to the instruction following the one that caused the condition.

C Signal Representation of S/370 Exceptions

S/370 exceptions and abends are mapped to C signals. Therefore, if both of the following are true:

- You have set the TRAP(ON) run-time option (LE/VSE condition handling is enabled)

- You do not request in the assembler user exit or in the ABPERC run-time option that any of the exceptions or abends be exempted from condition handling (ABPERC(NONE))

then you can apply C signal handling functions to S/370 exceptions and abends.

C signal representations for the following exceptions are provided in this section:

- For S/370 exceptions generated by the hardware or math library, see Table 32. Some of the exceptions listed in the table can be masked off for normal LE/VSE execution.
- For abends, see Table 33.

Table 32. Mapping of S/370 Exceptions to C Signals

Interrupt Code	Interrupt Code Description	C Signal Type
01	Operation exception	SIGILL
02	Privileged-operation exception	SIGILL
03	Execution exception	SIGILL
04	Protection exception	SIGSEGV
05	Addressing exception	SIGSEGV
06	Specification exception	SIGILL
07	Data exception	SIGFPE
08	Fixed-point overflow exception	n/a
09	Fixed-point divide exception	SIGFPE
10	Decimal-overflow exception	SIGFPE
11	Decimal-divide exception	SIGFPE
12	Exponent-overflow exception	SIGFPE
13	Exponent-underflow exception	n/a
14	Significance exception	n/a
15	Floating-point divide exception	SIGFPE

Table 33 lists the C signal type for abends that can occur under LE/VSE.

Table 33. Mapping of Abend Signals to C Signals

Message	Abend Description	C Signal Type
CEE3250	CICS-initiated abends	SIGABND
CEE3250	CICS user-initiated abends	SIGABND
CEE3321	VSE-initiated abends	SIGABND
CEE3322	VSE user-initiated abends	SIGABND
CEE3322	LE/VSE abends for severity 4 errors (U40xx)	n/a
CEE3322	LE/VSE-initiated abends	n/a

COBOL Condition Handling Semantics

COBOL native condition handling is very different from C's and PL/I's native condition handling.

COBOL provides some condition handling on a statement-by-statement basis: for example, the ON EXCEPTION phrase of the CALL statement, the ON EXCEPTION phrase of the INVOKE statement, and the ON SIZE ERROR phrase of the COMPUTE statement. For other conditions, COBOL generally reports the error. An assembler user exit is available for COBOL to specify events that should cause an abend.

For more information about user exits, see Chapter 25, “Using Run-Time User Exits,” on page 319. For a discussion of COBOL condition handling in an ILC application, see *LE/VSE Writing Interlanguage Communication Applications*. The following discussion applies to stacks comprised solely of COBOL programs.

If an exception occurs in a COBOL program, COBOL does nothing until every condition handler at every stack frame has been interrogated.

Once all stack frames have been visited, COBOL does the following:

1. Checks to see if the condition has a facility ID of IGZ (is a COBOL-specific condition). If not, COBOL percolates the condition to the LE/VSE condition manager.
2. Handles the condition based on its severity (see Table 28 on page 111 for an explanation of severity codes and their meaning under LE/VSE).

   If the condition severity is 1, a message describing the condition is issued to the destination specified in the MSGFILE run-time option, and processing resumes in the program in which the error occurred.

   If the severity is 2 or above, COBOL percolates the condition to the LE/VSE condition manager. The LE/VSE default action then takes place.

COBOL Condition Handling Examples

The following examples demonstrate how conditions are handled in LE/VSE if you do not use any LE/VSE callable services, and do not have any user-written condition handlers registered. The COBOLA program in Example Figure 40 on page 130 calls COBOLB in Example Figure 41 on page 130, which in turn calls the COBOLC program, in Example Figure 42 on page 131. A divide-by-zero condition occurs in COBOLC.

The divide-by-zero is enabled as a condition, so the condition step of LE/VSE condition handling is entered. There is no user-written condition handler registered for COBOLC or any of the other COBOL programs, so the condition is percolated through all of the stack frames. COBOL’s default action for the divide-by-zero condition is to percolate the condition to LE/VSE. The divide-by-zero condition has a severity of 3. LE/VSE’s default response to an unhandled severity 3 condition is to terminate the application and issue a message if TERMTHDACT(MSG) is specified.

```
CBL LIB,APOST,NODYNAM
      *Module/File Name: IGZTMLTA
      ************************************************************
      *                                                          *
      * Demonstrate a failing COBOL program with multiple active  *
      * routines on the stack. The call sequence is as follows:   *
      *                                                          *
      * COBOLA --> COBOLB --> COBOLC (which causes a zero divide) *
      *                                                          *
      ************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBOLA.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       1   Y  PIC 999 VALUE ZERO.
      *
       PROCEDURE DIVISION.
           DISPLAY 'In COBOLA.'.
           CALL 'COBOLB' USING Y.

           GOBACK.
```

Figure 40. COBOLA Program

Figure 41 calls COBOLC, and passes data to it.

```
CBL LIB,APOST,NOOPTIMIZE,NODYNAM
      * Module/File Name:  IGZTMLTB
      ************************************************************
      *                                                          *
      * Second routine called in the following call sequence:     *
      *                                                          *
      * COBOLA --> COBOLB --> COBOLC (which causes a zero divide) *
      *                                                          *
      ************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBOLB.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       1   X  PIC 999 VALUE ZERO.
       LINKAGE SECTION.
       1   Y  PIC 999.
      *
       PROCEDURE DIVISION USING Y.
           DISPLAY 'In COBOLB.'.
           MOVE Y TO X.
           CALL 'COBOLC' USING X.

           GOBACK.
```

Figure 41. COBOLB Program

Figure 42 generates a divide-by-zero condition. The divide-by-zero condition is percolated back to COBOLB, to COBOLA, and to LE/VSE default behavior.

```
CBL LIB,APOST,NODYNAM
      *Module/File Name: IGZTMLTC
      ***********************************************************
      *                                                         *
      * Third routine called in the following call sequence:    *
      *                                                         *
      * COBOLA --> COBOLB --> COBOLC (which causes a zero divide) *
      *                                                         *
      ***********************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. COBOLC.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       LINKAGE SECTION.
       1   Y  PIC 999.
      *
       PROCEDURE DIVISION USING Y.
           DISPLAY 'In COBOLC.'.
           COMPUTE Y = 1 / Y.

           GOBACK.
```

Figure 42. COBOLC Program

Restrictions about Resuming Execution after an IGZ Condition Occurs

You cannot *resume in place* after certain COBOL conditions (those with the facility ID IGZ) occur. If a user-written condition handler issued a *result_code* 10 (see “User-Written Condition Handler Interface using CEEHDLR” on page 138) without moving the resume cursor first, that would be a resume in place.

IGZ Condition of Severity 2 or Greater

A user-written condition handler cannot issue a resume in place after any IGZ condition of severity 2 or higher occurs. For example, if you encounter an error when trying to open a file, you cannot resume in place. You must either move the resume cursor and then resume, or percolate the condition.

COBOL STOP RUN Statement

There is a different constraint on resuming after a COBOL STOP RUN statement. When a STOP RUN is issued, Termination Imminent due to Stop (T_I_S) is raised (see “Processing the T_I_S Condition” on page 112 for more information about T_I_S). Therefore, you can respond to a STOP RUN by registering a user-written condition handler to recognize T_I_S.

This condition handler **cannot** call CEEMRCR with a 0 *type_of_move*, meaning move the resume cursor to the point in your program just after the STOP RUN statement. This violates the standard definition of a STOP RUN being the last statement to execute in the program in which it is coded. Assuming your program is a subroutine, you could issue a 1 *type_of_move*, meaning move the resume cursor to the call return point of the stack frame previous to the one of the program that issued the STOP RUN. You could also percolate the condition.

Reentering COBOL Programs after Stack Frame Collapse

A *stack frame collapse* occurs when the condition manager skips over one or more active routines and execution resumes in an earlier routine on the stack. This can occur due to either of the following:

- An explicit GOTO out of block issued from a C or PL/I routine
- Moving the resume cursor using the CEEMRCR callable service and requesting a resume

LE/VSE resets any intervening COBOL programs from an active to inactive state, provided they are the following:

- VS COBOL II programs compiled with the CMPR2 compiler option
- VS COBOL II programs compiled with NOCMPR2 that do not use *nested programs*
- COBOL/VSE programs compiled with the CMPR2 compiler option or
- COBOL/VSE programs compiled with NOCMPR2 that do not use the combination of the INITIAL attribute, nested programs, and files processing in the same compilation unit

After a stack frame collapse, the routines listed above can be reentered.

LE/VSE issues a warning message during stack frame collapse for each intervening COBOL program that does not adhere to the above restrictions. In addition, after the GOTO or resume is performed, any attempt to reenter these programs is diagnosed as an attempted recursive entry error.

Handling Fixed-Point and Decimal Overflow Conditions

The ON SIZE ERROR phrase continues to be invoked by COBOL to handle fixed-point and decimal overflow conditions, regardless of whether these conditions are enabled by LE/VSE.

PL/I Condition Handling Semantics

When an exception occurs in a PL/I routine, PL/I language semantics for handling the condition prevail. Therefore, the behavior of PL/I condition handling in applications consisting of only PL/I routines is unchanged under LE/VSE.

In PL/I, you handle all run-time conditions by writing ON-units. An ON-unit is a procedure that is established in a block when the ON statement for the ON-unit is run. The ON-unit itself runs when the specified condition in the ON statement is raised. The establishment of an ON-unit applies to all dynamically descendent (inherited from calling procedure) blocks of the block that established it; a condition occurring in a called procedure could result in an ON-unit being run in the caller.

This section provides a high-level view of how condition handling works if an exception occurs in a PL/I routine, and only PL/I routines are on the stack. For a more detailed explanation of PL/I condition handling, refer to *IBM PL/I for VSE/ESA Language Reference* For details about how PL/I condition handling works in an ILC application, see *LE/VSE Writing Interlanguage Communication Applications*.

PL/I Condition Handling Actions

Refer to Figure 43 on page 133 throughout the following summary of the steps taken to process a condition when there are only PL/I routines on the stack.

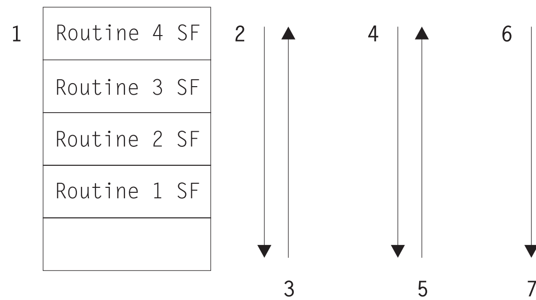


Figure 43. PL/I Condition Processing

1. Assume a condition such as CONVERSION, which is severity 3, occurs in routine 4.
2. If a debug tool is present, and the setting of the TEST run-time option indicates that it should be given control, it is invoked. See *LE/VSE Programming Reference* for information about the TEST run-time option.
3. If a debug tool is not invoked, or does not handle the condition, LE/VSE moves down the stack towards the earliest stack frame. If a PL/I ON-unit is established for the CONVERSION condition, it is given control.
4. If all stack frames have been visited and no ON CONVERSION unit was found, a message is issued. The condition is promoted to the ERROR condition if it meets any of the qualifications listed in “Promoting Conditions to the PL/I ERROR Condition.” Otherwise, the PL/I implicit action occurs.

   A CONVERSION condition would be promoted to ERROR.
5. If a debug tool is present, and the setting of the TEST run-time option indicates that it should be given control, it is invoked.
6. If a debug tool is not invoked, or does not handle the condition, the LE/VSE condition manager makes another pass of the stack, beginning in Routine 4 where the original condition occurred. If a PL/I ERROR ON-unit is established, it is invoked.
7. If either of the following occurs:
   - An ERROR ON-unit is found, but it does not issue a GOTO out of block.
   - No ERROR ON-unit is found.

   then the ERROR condition is promoted to T_I_U (Termination Imminent due to an Unhandled Condition). T_I_U maps to the PL/I FINISH condition. (See “Termination Imminent Step” on page 111 for a discussion of T_I_U.)
8. LE/VSE makes yet another pass of the stack, beginning in Routine 4 where the original condition occurred. If a PL/I FINISH ON-unit is established, it is invoked.
9. If all stack frames have been visited, and no FINISH ON-unit issued a GOTO out of block, then LE/VSE begins thread termination activities in response to the unhandled condition. Since a message was issued for the CONVERSION condition before it was promoted to the ERROR condition, no message is issued at this time.

Promoting Conditions to the PL/I ERROR Condition

PL/I promotes to the PL/I ERROR condition any PL/I condition for which the implicit action is to promote to the ERROR condition. The appropriate ONCODE is used. See *IBM PL/I for VSE/ESA Language Reference* for details.

Mapping Non-PL/I Conditions to PL/I Conditions

Some non-PL/I conditions map directly to PL/I conditions:

- The LE/VSE conditions listed in the first column below map directly to the PL/I conditions in the second column.

  Decimal divide
  ZERODIVIDE

  Decimal overflow
  FIXEDOVERFLOW

  Exponent overflow
  OVERFLOW

  Exponent underflow
  UNDERFLOW

  Fixed-point divide
  ZERODIVIDE

  Fixed-point overflow
  FIXEDOVERFLOW

  Floating-point divide
  ZERODIVIDE

  These LE/VSE conditions map directly to the PL/I conditions. They are detected by the hardware and are normally represented by condition tokens with a facility ID of CEE when raised. They are represented by an IBM condition token only when signaled by the PL/I SIGNAL statement.

- The following conditions map directly to ERROR:
  - An LE/VSE condition of severity 2, 3, or 4 that does not map to one of the PL/I conditions listed above.

    For these conditions, an established ERROR ON-unit is run on the first pass of the stack. In general, the ONCODE is 9999. Some LE/VSE conditions that map to ERROR, however, are represented by an ONCODE other than 9999. Examples are some of the conditions raised by the LE/VSE math services.
  - Any other condition of severity 2, 3, or 4.

    For these conditions, an established ERROR ON-unit is run on the first pass of the stack. The ONCODE is 9999.

Additional PL/I Condition Handling Considerations

Keep the following additional PL/I condition handling considerations in mind:

- Non-PL/I conditions of severity 0 or 1 are not promoted to ERROR.
- Promoting any non-PL/I condition to a PL/I condition is prohibited.
- Raising a PL/I condition through the use of the CEESGL callable service is prohibited.
- Issuing a call to CEEMRCR from within a PL/I ON-unit in order to move the resume cursor is prohibited. You can move the resume cursor by using CEEMRCR from within a LE/VSE user-written condition handler, however.

PL/I Condition Handling Example

```
*PROCESS MACRO;
 /*Module/File Name: IBMDIVZ
 /*******************************************************************/
 /*                                                                 */
 /* PL/I Condition Handling Functions:                              */
 /*                  : Establish ZERODIVIDE ON-unit                 */
 /*                  : GO TO out of ZERODIVIDE ON-unit              */
 /*                  : PL/I Normal return from ZERODIVIDE ON-unit   */
 /*                  : Revert ZERODIVIDE ON-unit                    */
 /*                  : PL/I System action on ZERODIVIDE condition   */
 /*                                                                 */
 /* 1. This example establishes a ZERODIVIDE ON-unit.               */
 /* 2. A sub-program, sdivide, is called and causes a ZERODIVIDE    */
 /*    condition to occur.                                          */
 /* 3. The ZERODIVIDE ON-unit is entered. A GOTO out of the ON-unit */
 /*    is processed.  The program resumes at the label              */
 /*    "after_1st_zerodivide".                                      */
 /* 4. A new ZERODIVIDE ON-unit is established and it overrides the */
 /*    current established ZERODIVIDE ON-unit.                      */
 /* 5. The subroutine sdivide is called a second time.              */
 /* 6. The newly established ZERODIVIDE ON-unit is entered. A GOTO  */
 /*    is not executed, and the program resumes at the location     */
 /*    following the instruction that caused the condition. This    */
 /*    is the PL/I normal return action for the ZERODIVIDE condition. */
 /* 7. The established ZERODIVIDE ON-unit is canceled by executing  */
 /*    the REVERT ZERODIVIDE statement.                             */
 /* 8. Sdivide is called a third time.  Because there is no         */
 /*    ZERODIVIDE ON-unit established, the PL/I implicit action     */
 /*    is executed. Namely, the ERROR condition is raised and the   */
 /*    program is terminated.                                       */
 /**                                                                */
 /*******************************************************************/

 CEPLCND: Proc Options(Main);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;
    dcl in_zdiv_ou1 char (1), in_zdiv_ou2 char(1),fell_thru char(1);
    in_zdiv_ou1 = 'N';
    in_zdiv_ou2 = 'N';
    fell_thru = 'N';
    /****************************************************************/
    /* A ZERODIVIDE ON-unit is established when control reaches the */
    /* ON statement.                                                */
    /****************************************************************/
     on zerodivide begin;
        in_zdiv_ou1 = 'Y';
        go to after_1st_zerodivide;
        end;

    /****************************************************************/
    /* The first call to sdivide will result in the ZERODIVIDE      */
    /* condition being raised. The preceding established ON-unit    */
    /* gets control.  Due to a GO TO out of the ON-unit, execution  */
    /* resumes immediately at label after_1st_zerodivide. This is   */
    /* verified by checking that the flow of control did not resume */
    /* at the instruction following the ZERODIVIDE condition.       */
    /****************************************************************/
```

Figure 44. PL/I Condition Handling Example (Part 1 of 2)

```
   call sdivide;
after_1st_zerodivide:
   if (fell_thru = 'Y') then do;
      put skip list ('Error in flow of control after'
         || ' the first call to sdivide. ');
      end;
   /**************************************************************/
   /* A new ZERODIVIDE ON-unit is established when control       */
   /* reaches the following ON ZERODIVIDE statement.             */
   /**************************************************************/
   on zerodivide begin;
      in_zdiv_ou2 = 'Y';
      end;

   /**************************************************************/
   /* Subroutine sdivide is called a second time to raise the    */
   /* ZERODIVIDE condition.  Control enters the established      */
   /* ZERODIVIDE ON-unit.  On exit from the preceding zerodivide */
   /* ON-unit, control returns to the instruction following the  */
   /* divide by zero in subroutine SDIVIDE.  A check is made to  */
   /* detect if control flowed to the instruction following the  */
   /* one that caused the zerodivide condition to be raised.     */
   /**************************************************************/
   call sdivide;
   if (fell_thru = 'N') then do;
      put skip list
         ('Error in flow of control after second call to cepldiv. ');
      end;
   /**************************************************************/
   /* The ZERODIVIDE  ON-unit is canceled by action of the       */
   /* REVERT statement.                                          */
   /**************************************************************/
   revert zerodivide;
   if (in_zdiv_ou1 = 'N' | in_zdiv_ou2 = 'N') then
      put skip list ('Error in flow of control to ON-units');
   else  do;
      put skip list ('The PL/I condition handling example'
         || ' will terminate with PL/I message IBM0301');

      /***********************************************************/
      /* Sdivide is called for the third and final time. Because */
      /* there are no established ON-units, the implicit action  */
      /* for ZERODIVIDE takes place.                             */
      /***********************************************************/
      call sdivide;
      put skip list ('Error in flow of control after third'
         || ' call to sdivide. ');
      end;

/*****************************************************************/
/* The sdivide subroutine causes a ZERODIVIDE condition.         */
/*****************************************************************/
sdivide: proc;
     dcl int fixed bin (15,0);
     dcl int_2 fixed bin (15,0) init(5);
     dcl int_3 fixed bin (15,0) init(0);
     int = int_2 / int_3;
     fell_thru = 'Y';
end sdivide;

End CEPLCND;
```

Figure 44. PL/I Condition Handling Example (Part 2 of 2)

Chapter 13. Coding a User-Written Condition Handler

This chapter describes how you can code a user-written condition handling routine and provides examples for LE/VSE-conforming HLLs.

Understanding the Basics

Your user-written condition handler can test for the occurrence of a particular condition by coding a 12-byte condition token or by coding a symbolic feedback code. You can use CEEHDLR when coding your user-written condition handler to register the condition handler. For information about using CEEHDLR, see “User-Written Condition Handler Interface using CEEHDLR” on page 138.

User-written condition handlers cannot be written in PL/I, be registered from a PL/I routine, or use PL/I pseudovariables or condition handling built-in functions. However, PL/I ON-units can call programs written in C or COBOL that can register a user-written condition handler. Similarly, user-written condition handlers can call PL/I routines, which can then establish ON-units. ON-units can make calls to any LE/VSE services except CEEMRCR.

Also, the USRHDLR run-time option enables you to register a user-written condition handler at stack frame 0 without having to recompile your application to include a call to CEEHDLR. This is particularly useful in supporting PL/I applications that are unable to directly call CEEHDLR.

Condition handlers written in COBOL must be compiled with the COBOL/VSE compiler.

Nested conditions can be used in your routine as long as the language your routine is written in allows it to be recursively entered. You should design the routine to handle specific conditions rather than designing the routine to handle a wide variety of conditions. You should also code the condition handling routine to respond to the original condition on the first pass of the stack, rather than coding a routine to handle T_I_U on the second pass of the stack. This helps ensure that the handling that you perform addresses the original condition. The more specific the condition is that you design the handler for, the more precise the fix can be.

Types of Conditions You Can Handle

A user-written condition handler can, in general, intercept and process any condition, regardless of the language of the routine in which the condition occurred. This means that you can code a user-written condition handler to respond to condition tokens with any of the following facility IDs:

- CEE, representing LE/VSE conditions
- EDC, representing C conditions
- IGZ, representing COBOL conditions
- IBM, representing PL/I conditions

In general, your user-written condition handler can use any of the LE/VSE condition handling services. Specific exceptions follow:

- The ways in which you can resume after an IGZ condition of severity 2 or above are restricted. See "Restrictions about Resuming Execution after an IGZ Condition Occurs" on page 131 for details.
- If an IBM condition of severity 2 or above was raised, then you cannot issue a resume without first moving the resume cursor.

  This restriction does not apply to IBM conditions of severity 0 or 1, or any IBM conditions signaled using the PL/I SIGNAL statement.
- You cannot promote any condition to a condition with a facility ID of IBM (one that belongs to PL/I). You can promote IBM conditions to conditions with facility IDs of CEE, EDC, or IGZ.

For more information on coding user-written condition handlers to respond to conditions of different facility IDs, see "Using Symbolic Feedback Codes" on page 175.

User-Written Condition Handler Interface using CEEHDLR

Use CEEHDLR to register a user-written condition handler. See *LE/VSE Programming Reference* for more information about CEEHDLR.

User-written condition handlers are automatically unregistered when the stack frame they're associated with is removed from the stack due to a return, GOTO out of block, or a move of the resume cursor. You can, however, call CEEHDLU to explicitly unregister a user-written condition handler. See *LE/VSE Programming Reference* for more information about CEEHDLU.

Recursion is allowed if a handler is registered within a handler, and nested conditions are allowed.

It is invalid to promote a condition without returning a new condition token. You cannot promote a condition to a PL/I condition.

Syntax

```
►►──condition_handler──(──c_ctok──,──token──,──result_code──,──new_condition──)──────►◄
```

c_ctok (input)
: A 12-byte condition token that identifies the current condition being processed. LE/VSE uses this parameter to tell your condition handler what condition has occurred.

token (input)
: A 4-byte integer that specifies the token you passed into LE/VSE when this condition handler was registered by a call to the CEEHDLR callable service.

result_code (output)
: A 4-byte integer that contains instructions about responses the user-written condition handler wants LE/VSE to make when processing the condition.

  result_code is passed by reference. The following responses are valid:

Response	Result_Code Value	Action
resume	10	Resume at the resume cursor (condition has been handled).

Response	Result_Code Value	Action
percolate	20	Percolate to the next condition handler. If a *result_code* is not explicitly set by a handler, this is the default *result_code*.
	21	Percolate to the first user-written condition handler for the stack frame that is before the one to which the handle cursor points. This can skip a language-specific condition handler for this stack frame as well as the remaining user-written condition handlers in the queue for this stack frame.
promote	30	Promote to the next condition handler.
	31	Promote to the stack frame before the one to which the handle cursor points. This can skip a language-specific condition handler for this stack frame as well as any remaining user-written condition handler in the queue at this stack frame.
	32	Promote and restart condition handling at the first condition handler of the stack frame of the handle cursor.

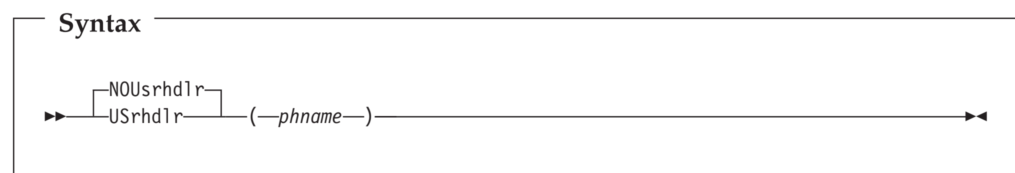
If *result_code* is not explicitly set by the handler, the default response is `Value=20, Percolate to the next condition handler.`

new_condition (output)
A 12-byte condition token that represents the promoted condition. This field is used only for *result_code* values of 30, 31, and 32, denoting **promote**.

Registering a User-Written Condition Handler using USRHDLR

Use the USRHDLR run-time option to register a user-written condition handler at stack frame 0. The condition handler specified is invoked after the default HLL condition handler for the main program, but before the HLL condition handler for stack frame 0. The condition handler percolated or promoted by the user-written condition handler registered by USRHDLR, is not passed to any other condition handler.

Syntax



NOUsrhdlr
Specifies that no user-written condition handler is registered.

USrhdlr
Specifies that a user-written condition handler is registered at stack frame 0.

phname
The entry point name of a phase that contains the user-written condition handler to be registered at stack frame 0.

The condition handler registered by the USRHDLR run-time option can return any of the result codes allowed for a condition handler registered with the CEEHDLR callable service.

For more information about using the USRHDLR run-time option, refer to the *LE/VSE Programming Reference*.

Nested Conditions

A *nested condition* is one that occurs within a C signal handler, PL/I ON-unit, or user-written condition handler invoked to handle a condition. When conditions occur during the condition handling process, the handling of the original condition is suspended and further action is taken based on the state of the condition handling.

The DEPTHCONDLMT run-time option indicates whether nested conditions are permitted while your application runs. If you specify DEPTHCONDLMT(1), handling of the initial condition is allowed, but any additional nested condition causes your application to abend. If you specify DEPTHCONDLMT(0), an unlimited number of nested conditions is permitted. If you specify some other integer value for DEPTHCONDLMT, LE/VSE allows handling of the initial condition plus additional levels of nested conditions before your application abends (see *LE/VSE Programming Reference* for more information).

If a nested condition is allowed within a user-written condition handler, LE/VSE begins handling the most recently raised condition. After the most recently raised condition is properly handled, execution begins at the instruction pointed to by the resume cursor, the instruction following the point where the condition occurred. If a user-written condition handler is registered using CEEHDLR within another user condition handler, nested conditions are handled by the most recently registered condition handler.

If any HLL or user-written condition handler moves the resume cursor closer to the oldest stack frame both conditions are considered handled. The application resumes running at the instruction pointed to by the resume cursor. The resume cursor can be moved using the CEEMRCR callable service, or by language constructs such as GOTO.

Nested Conditions in Applications Containing a COBOL Program

You must take special care when dealing with nested conditions in ILC applications. For example, the following scenario can cause your application to abend:

1. A nested condition occurs within a COBOL user-written condition handler (COBOL_UHDLR).
2. The COBOL user-written condition handler calls another user-written condition handler established using CEEHDLR to handle the nested condition.
3. The user-written condition handler percolates the condition.

In this scenario, the condition can be percolated back to the stack frame where the original condition occurred. Since condition handling actions for the routine where the condition originally occurred include calling COBOL_UHDLR, COBOL_UHDLR can be recursively entered. This is not permitted under COBOL, and your application abends.

A rule of thumb is to ensure that COBOL user-written condition handlers that call other user-written handlers do not regain control.

Using LE/VSE Condition Handling with Nested COBOL Programs

If your application contains both nested COBOL programs and calls to LE/VSE condition handling services, keep the following in mind:

- Do not call CEEHDLR from a nested COBOL program.
- Do not call CEEMRCR with a 1 *type_of_move* from a user handler associated with a stack frame that was called by a nested COBOL program. In Figure 45, Program A calls nested Program B. Program B calls Program C, which registers a user-written condition handler, UWCHC. UWCHC cannot call CEEMRCR with a 1 *type_of_move*, which would move the resume cursor back to nested Program B.

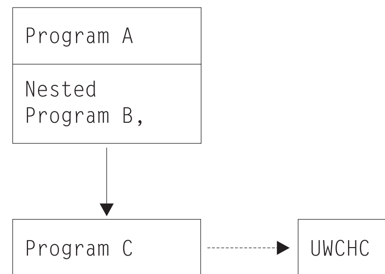


Figure 45. Restricted type_of_move If COBOL Nested Programs Are Present

Examples with a Registered User-Written Condition Handler

This section contains C, COBOL, and assembler examples in which user-written condition handlers are registered to respond to specific conditions that might occur in an application.

- In "Handling a Divide-by-Zero Condition in C or COBOL," C and COBOL call CEEHDLR and CEEMRCR to handle a divide-by-zero condition.
- In "Handling an Out-of-Storage Condition in C or COBOL" on page 148, C and COBOL call CEEHDLR and CEEMRCR to handle an out-of-storage condition.
- In "Signaling and Handling a Condition in a C Routine" on page 158, C calls CEEHDLR, CEEGQDT, and CEEMRCR to respond to a signaled condition.
- In "Handling a Divide-by-Zero Condition in a COBOL Program" on page 160, COBOL calls CEEHDLR, CEE5GRN, and CEEMOUT to respond to the significance condition (which was enabled using CEE5SPM).
- In "Handling a Program Check in an Assembler Routine" on page 165, assembler calls CEEHDLR to register a condition handler that responds to a program check.

You cannot register PL/I routines as user-written condition handlers using CEEHDLR. However, user-written condition handlers written in C or COBOL can call PL/I routines.

Handling a Divide-by-Zero Condition in C or COBOL

Figure 46 on page 142 and the following examples provide an illustration of how user-written condition handlers can handle conditions such as a divide-by-zero in a C or COBOL application. In Figure 47 on page 143 (for C) or Figure 48 on page 145 (for COBOL), the main routine EXCOND calls CEEHDLR to register the USRHDLR user-written condition handler (Figure 50 on page 147). EXCOND then calls the

DIVZERO routine (Figure 49 on page 146), in which a divide-by-zero exception occurs.

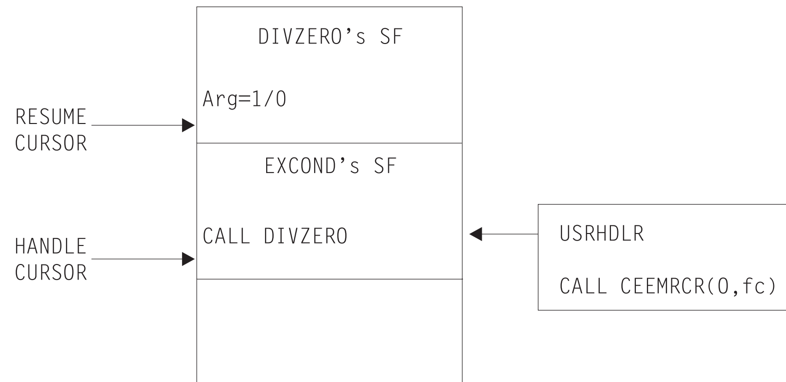


Figure 46. Handle and Resume Cursor Movement as a Condition Is Handled

Divide-by-zero is enabled as a condition in the following steps:

1. The handle cursor, which first points at DIVZERO's stack frame, moves down the stack to the USRHDLR condition handler, the first user-written condition handler established to handle conditions for the main routine's stack frame.
2. For divide-by-zero conditions, USRHDLR issues a call to CEEMRCR (Move Resume Cursor Relative to Handle Cursor) with a 0 *type_of_move*, meaning move the resume cursor to the call return point of the stack frame associated with the handle cursor. (The call return point is the next instruction after the call to the DIVZERO routine.)
3. Execution resumes in EXCOND at this point. A divide-by-zero condition is the only type of program interrupt for which USRHDLR causes a resume.
4. All other program interrupts are percolated to the next condition handler on the stack.

Note: For simplicity, the examples shown below do not include calls to some LE/VSE services that might otherwise be useful for handling conditions in your application. For example, you might code in the USRHDLR routine a call to the CEE5GRN callable service in order to get the name of the routine that incurred the condition.

C Handling a Divide-by-Zero Condition

Figure 47 is the C routine that performs the tasks discussed above.

```
/*Module/File Name:  EDCDIVZ  */
/**********************************************************/
/*                                                        */
/*  MAIN                 .-> DIVZERO                      */
/*  - register handler   |   - force a divide-by-zero     */
/*  - call DIVZERO     --'                                */
/*  ==> "resume point"                                    */
/*  - unregister handler                                  */
/*                         USRHDLR:                       */
/*                         - if divide-by-zero            */
/*                           - move resume cursor         */
/*                           - resume at "resume point"   */
/*                                                        */
/**********************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

void usrhdlr(_FEEDBACK *, _INT4 *, _INT4 *, _FEEDBACK *);

void divzero(int);

int main(void) {

 _FEEDBACK fc;
 _INT4 divisor;
 _INT4 token;
 _ENTRY pgmptr;

 /* Register a user-written condition handler.            */
 pgmptr.address = (_POINTER)&usrhdlr;
 pgmptr.nesting = NULL;
 token = 97;
 CEEHDLR (&pgmptr, &token, &fc);
 if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
    printf( "CEEHDLR failed with message number %d\n",
       fc.tok_msgno);
    exit(99);
   }
 printf("MAIN: Registered USRHDLR.\n");

 /* Call DIVZERO to divide by zero and drive USRHDLR      */
 divisor = 0;
 divzero(divisor);
 printf("MAIN: Resumption after DIVZERO.\n");
```

Figure 47. EXCOND Routine (C) (Part 1 of 2)

```
 /* Unregister the user condition handler.                */
 CEEHDLU (&pgmptr, &fc);
 if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
    printf( "CEEHDLU failed with message number %d\n",
       fc.tok_msgno);
    exit(99);
   }

 printf("MAIN: Unregistered USRHDLR.\n");
} /* end main */

void divzero(int arg) {
 printf("  DIVZERO: Starting.\n");
 arg = 1 / arg;
 printf("  DIVZERO: Returning to its caller.\n");
} /* end divzero */

/*********************************************************/
/* usrhdlr will handle DIVIDE-BY-ZERO conditions...      */
/*    all others will be percolated.                     */
/*********************************************************/

void usrhdlr(_FEEDBACK *cond,_INT4 *input_token,
             _INT4 *result, _FEEDBACK *new_cond)
 {
  _INT4 move_type_0 = 0;
  _INT4 move_type_1 = 1;
  _FEEDBACK feedback;

  /* values for handling the conditions */
  #define resume      10
  #define percolate   20
  #define promote     30
  #define promote_sf  31

  printf(">>> USRHDLR: Entered User Handler \n");
  printf(">>> passed token value is %d\n",*input_token);

  /* check if the DIVIDE-BY-ZERO message (0C9) */
  if (cond->tok_msgno == 3209) {
      CEEMRCR (&move_type_0, &feedback);
      if ( _FBCHECK ( feedback , CEE000 ) != 0 ) {
         printf( "CEEMRCR failed with message number %d\n",
           feedback.tok_msgno);
         exit(99);
        }
      *result = resume;
      printf(">>> USRHDLR: Resuming execution\n");
     }
  else {                      /* not DIVIDE-BY-ZERO */
        *result = percolate;
        printf(">>> USRHDLR: Percolating it\n");
       }
} /* end usrhdlr */
```

Figure 47. EXCOND Routine (C) (Part 2 of 2)

COBOL Handling a Divide-by-Zero Condition

Figure 48 registers a user-written condition handler, calls the DIVZERO subroutine, and unregisters the condition handler on return from the subroutine.

```
CBL LIB,APOST,NODYNAM,NOOPT
      *Module/File Name: IGZTDIVZ
      *****************************************************
      *                                                   *
      *  EXCOND               .-> DIVZERO                 *
      *  - register handler   |   - force a divide-by-zero *
      *  - call DIVZERO     --'                           *
      *  ==> 'resume point'                               *
      *  - unregister handler                             *
      *                       USRHDLR                     *
      *                       - if divide-by-zero, then:  *
      *                         - move resume cursor      *
      *                         - resume at 'resume point' *
      *                                                   *
      *****************************************************

       IDENTIFICATION DIVISION.
       PROGRAM-ID.    EXCOND.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77  DIVISOR         PIC S9(9) BINARY.
      **
      **  Declarations for condition handling
      **
       77  TOKEN           PIC X(4).
       77  PGMPTR          USAGE IS PROCEDURE-POINTER.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity        PIC S9(4) BINARY.
                   04  Msg-No          PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                           REDEFINES Case-1-Condition-ID.
                   04  Class-Code      PIC S9(4) BINARY.
                   04  Cause-Code      PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info        PIC S9(9) BINARY.

       PROCEDURE DIVISION.
       PARA-CND01A.
      ************************************************
      ** Register a user-written condition handler. **
      ************************************************
           SET PGMPTR TO ENTRY 'USRHDLR'.
           MOVE ZERO TO TOKEN.
           CALL 'CEEHDLR' USING PGMPTR TOKEN FC.
           IF CEE000 of FC  THEN
               DISPLAY 'EXCOND: REGISTERED USRHDLR.'
           ELSE
               DISPLAY 'CEEHDLR failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.
```

Figure 48. EXCOND Routine (COBOL) (Part 1 of 2)

```
*************************************************************
** Call DIVZERO to force a divide-by-zero and drive USRHDLR **
*************************************************************
     MOVE 00 TO DIVISOR.
     CALL 'DIVZERO' USING DIVISOR.
     DISPLAY 'EXCOND: RESUMED AFTER DIVZERO.'.
***************************************************
** Unregister the user-written condition handler.**
***************************************************
     CALL 'CEEHDLU' USING PGMPTR FC.
     IF CEE000 of FC  THEN
         DISPLAY 'EXCOND: UNREGISTERED USRHDLR.'
     ELSE
         DISPLAY 'CEEHDLU failed with msg '
             Msg-No of FC UPON CONSOLE
         STOP RUN
     END-IF.

     GOBACK.
 END PROGRAM EXCOND.
```

Figure 48. EXCOND Routine (COBOL) (Part 2 of 2)

Figure 49 is a subroutine that generates the divide-by-zero condition.

```
CBL LIB,APOST,NODYNAM,NOOPT
      *Module/File Name: IGZTDIVS
       IDENTIFICATION DIVISION.
       PROGRAM-ID. DIVZERO.

       DATA DIVISION.
       LINKAGE SECTION.
       01  ARG     PIC S9(9) BINARY.

       PROCEDURE DIVISION USING ARG.
           DISPLAY '  DIVZERO: STARTING.'.
           COMPUTE ARG = 1 / ARG.
           DISPLAY '  DIVZERO: RETURNING TO ITS CALLER.'.

           GOBACK.
       END PROGRAM DIVZERO.
```

Figure 49. DIVZERO Routine (COBOL)

Figure 50 is the user-written condition handler registered by EXCOND to handle the divide-by-zero condition. When the divide-by-zero condition arises, USRHDLR calls CEEMRCR with a 0 *type of move*. Doing so moves the resume cursor to the point in EXCOND after the call to DIVZERO.

```
CBL LIB,APOST
      *Module/File Name: IGZTDIVU
      *****************************************************
      *                                                   *
      * USRHDLR                                           *
      *                                                   *
      *****************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. USRHDLR.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  MISC-VARIABLES.
           02  MOVE-TYPE-0     PIC S9(9) BINARY VALUE ZERO.
           02  MOVE-TYPE-1     PIC S9(9) BINARY VALUE 1.
       01  FEEDBACK.
           02  FB-SEVERITY     PIC 9(4)  BINARY.
           02  FB-DETAIL       PIC X(10).
      *
       LINKAGE SECTION.
      *****************************************************
      *                                                   *
      *  Note:  the symbolic names of the condition tokens *
      *  for S/370 program interrupt codes 0C1 thru 0CF    *
      *  are CEE341 through CEE34F                         *
      *                                                   *
      *****************************************************
       01  TOKEN           PIC X(4).

       01  RESULT-CODE     PIC S9(9) BINARY.
           88 RESUME           VALUE +10.
           88 PERCOLATE        VALUE +20.
           88 PERC-SF          VALUE +21.
           88 PROMOTE          VALUE +30.
           88 PROMOTE-SF       VALUE +31.

       01  CURRENT-CONDITION.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity        PIC S9(4) BINARY.
                   04  Msg-No          PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code      PIC S9(4) BINARY.
                   04  Cause-Code      PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info        PIC S9(9) BINARY.
```

Figure 50. USRHDLR Routine (COBOL) (Part 1 of 2)

```
 01  NEW-CONDITION.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity       PIC S9(4) BINARY.
             04  Msg-No         PIC S9(4) BINARY.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code     PIC S9(4) BINARY.
             04  Cause-Code     PIC S9(4) BINARY.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info       PIC S9(9) BINARY.

 PROCEDURE DIVISION USING CURRENT-CONDITION TOKEN
                          RESULT-CODE NEW-CONDITION.

     DISPLAY '>>> USRHDLR: Entered User Condition Handler '.
     IF CEE349 of CURRENT-CONDITION THEN
**********************************************************
*     Expected condition, divide by zero, occurred...    *
*     move resume cursor to stack frame which registered *
*     the handler, and resume execution at that point.   *
**********************************************************
         CALL 'CEEMRCR' USING MOVE-TYPE-0 FEEDBACK
         SET RESUME TO TRUE
         DISPLAY '>>> USRHDLR: Resuming execution'
     ELSE
**********************************************************
*        UNexpected condition encountered.. percolate it!*
**********************************************************
         SET PERCOLATE TO TRUE
         DISPLAY '>>> USRHDLR: Percolating it'
     END-IF.

     GOBACK.
 END PROGRAM USRHDLR.
```

Figure 50. USRHDLR Routine (COBOL) (Part 2 of 2)

Handling an Out-of-Storage Condition in C or COBOL

You can use the LE/VSE condition handling services to resolve an out of storage condition in your application. In the user-written condition handler examples that follow, CEEGTST and CEECZST are used to get and reallocate heap storage. CEEMRCR is also used to handle an out-of-storage condition in a user subroutine, and allow the subroutine to be invoked again. To see the user code that corresponds to this scenario, see:

- See the Examples on 150 and 152, for C
- See the Examples on 153, 155, and 157, for COBOL

The out-of-storage condition is handled as follows:

1. The out-of-storage condition arises in your function (C) or subroutine (COBOL), and LE/VSE gives control to the user-written condition handler you have registered through CEEHDLR for the out-of-storage condition.
2. The condition handler detects the out-of-storage condition and calls CEEMRCR to set the resume cursor to resume execution at the return address of your subroutine call.
3. On return from the user condition handler, your main program regains control as if your subroutine has actually run.

4. The main program tests a completion indicator and discovers that the subroutine did not actually complete.
5. Your program then recognizes that it has been invoked with insufficient storage for maximum efficiency, and frees some previously allocated storage.
6. The subroutine is invoked a second time and completes successfully.

See *LE/VSE Programming Reference* for syntax of all LE/VSE condition handling services.

C Examples Using CEEHDLR, CEEGTST, CEECZST, and CEEMRCR

Figure 51 calls CEEHDLR to register a user-written condition handler for the out-of-storage condition, calls CEEGTST to allocate heap storage, and calls CEECZST to alter the size of the heap storage requested.

```
/*Module/File Name:  EDCOOSR  */
 /*********************************************************/
 /*                                                       */
 /* Function  : CEEHDLR - Register user condition handler */
 /*           : CEEGTST - Get Heap Storage                */
 /*           : CEECZST - Change the size of heap element */
 /*                                                       */
 /* 1. A user condition handler CECNDHD is registered.    */
 /* 2. A large amount of HEAP storage is allocated.       */
 /* 3. An inline function, sub(), is called which         */
 /*    requires a large amount of storage. It is not      */
 /*    known whether the storage for sub() is             */
 /*    available during this run of the application.      */
 /* 4. If sufficient storage for sub() is not avail-      */
 /*    able, a storage condition is generated by LE/VSE.  */
 /* 5. CECNDHD get control and sets resume at the         */
 /*    next instruction following the call to sub().      */
 /* 6. A test for completion of sub() is made after       */
 /*    the function call. If sub() did not complete,      */
 /*    a large amount of storage is freed, and sub()      */
 /*    is invoked a second time.                          */
 /* 7. sub() runs successfully once it has enough         */
 /*    storage available.                                 */
 /*                                                       */
 /*    Note:  In order for this example to complete       */
 /*    successfully, the FREE suboption of the HEAP       */
 /*    runtime option must be in effect.                  */
 /*                                                       */
 /*********************************************************/
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>

#define BIGSTOR 300000
#define BIGINDX BIGSTOR-1

 void CECNDHD(_FEEDBACK *, _INT4 *, _INT4 *, _FEEDBACK *);

 char *sub( );
 void main ()
 {
   _FEEDBACK feedback;
   _ENTRY pgmptr;
   _POINTER addrss;
   _INT4 token;
   _INT4 hpsize;
   _INT4 heapid;
   _INT4 newsize;
   char *RAN;
```

Figure 51. C Example of a main() Routine That Calls a Function and Registers a Condition Handler for an Out-of-Storage Condition (Part 1 of 2)

```
/**********************************************************/
/* Call CEEHDLR to register user condition handler CECNDHD */
/**********************************************************/
  pgmptr.address = (_POINTER)&CECNDHD;
  pgmptr.nesting = NULL;
  token = 97;
  CEEHDLR(&pgmptr, &token, &feedback);
  if ( _FBCHECK ( feedback , CEE000 ) != 0 )
     printf( "CEEHDLR failed with message number %d\n",
             feedback.tok_msgno);
  else
     printf( "Condition handler registered\n" );
/**********************************************************/
/* Call subroutine sub( ). When sub becomes active, an out */
/* of storage condition arises if the region is too small  */
/**********************************************************/
 heapid = 0;
 hpsize = BIGSTOR;
 CEEGTST ( &heapid , &hpsize , &addrss , &feedback );
 if ( _FBCHECK ( feedback , CEE000 ) != 0 )
    printf("CEEGTST failed with message number %d\n",
            feedback.tok_msgno);
 RAN = sub ( );
 if (RAN != "r")
   {
  /*******************************************************/
  /* If Sub did not run, reduce the size of allocated    */
  /*  storage and call Sub a second time.                */
  /*******************************************************/
     newsize = 2000;
     CEECZST ( &addrss, &newsize, &feedback );
     if ( _FBCHECK ( feedback , CEE000 ) != 0 )
        printf( "CEECZST failed with message number %d\n",
                feedback.tok_msgno);
     printf("Subroutine is called for the 2nd time\n");
     RAN = sub ( );
     printf("Subroutine %.can sucessfully\n", *RAN);
   };
} /* end of main */

char *sub( )
{
  char w2[BIGSTOR];
  w2[BIGINDX] = 'B';
  return( "r" );
} /* end of sub */
```

Figure 51. C Example of a main() Routine That Calls a Function and Registers a Condition Handler for an Out-of-Storage Condition (Part 2 of 2)

When any condition occurs in the C program (Figure 51), the user-written condition handler (Figure 52) receives control and tests for the out-of-storage condition. If the out-of-storage condition has occurred, then the example calls CEEMRCR to return to the instruction in the C main after the call to function `sub()`.

```
/*Module/File Name:  EDCOOSH  */
 /*********************************************************/
 /*                                                       */
 /* Function  : CEEMRCR - Move resume cursor relative     */
 /*                       to handle cursor.               */
 /*                                                       */
 /* CECNDHD is a user condition handler that is registered */
 /* by the program CECNDXP.  CECNDHD gets control from the */
 /* condition manager and tests for the STORAGE CONDITION. */
 /* If a STORAGE CONDITION is detected, the resume cursor  */
 /* is moved so that control is returned to the caller of  */
 /* the routine encountering the STORAGE CONDITION.        */
 /*                                                       */
 /*********************************************************/
 #include <stdio.h>
 #include <string.h>
 #include <leawi.h>
 #include <ceeedcct.h>
 #define RESUME 10
 #define PERCOLATE 20
 #define PROMOTE 30
 #define PROMOTE_STACK_FRAME 31

 void CECNDHD (_FEEDBACK *cond, _INT4 *input_token,
               _INT4 *result, _FEEDBACK *new_cond)

{
 _FEEDBACK feedback;
 _INT4 movetyp;

 /*********************************************************/
 /* Determine if entry was for OUT OF STORAGE condition.   */
 /*********************************************************/
 if ( _FBCHECK (*cond , CEE0PD) == 0 )
  {
    printf("CESUBXP not run because of storage condition.\n");
    /****************************************************/
    /* Call CEEMRCR to move resume cursor.              */
    /****************************************************/
    movetyp = 0;
    CEEMRCR ( &movetyp , &feedback );
    if ( _FBCHECK ( feedback , CEE000) != 0 )
     {
       *result = PERCOLATE;
     }
    else
     {
       *result = RESUME;
     }
  }
 else
  {
   /*****************************************************/
   /* Percolate all conditions except for OUT OF STORAGE  */
   /*****************************************************/
   *result = PERCOLATE;
  }
}
```

Figure 52. C User-Written Condition Handler Registered for the Out-of-Storage Condition

COBOL Examples Using CEEHDLR, CEEGTST, CEECZST, and CEEMRCR

The Figure 51 calls CEEHDLR to register a user-written condition handler for the out-of-storage condition, calls CEEGTST to allocate heap storage, and calls CEECZST to alter the size of the heap storage requested.

```
CBL LIB,APOST,NODYNAM
      *Module/File Name: IGZTOOSR
      *************************************************************
      *  CECNDXP - Call the following LE services:                *
      *             : CEEHDLR - Register user condition handler   *
      *             : CEEGTST - Get Heap Storage                  *
      *             : CEECZST - Change the size of heap element   *
      *                                                           *
      *    1. A user condition handler CECNDHD is registered.     *
      *    2. A large amount of HEAP storage is allocated.        *
      *    3. A subroutine, CESUBXP is called which is known to   *
      *       require a large amount of storage. It is not known  *
      *       whether the storage for CESUBXP is available during *
      *       this run of the application.                        *
      *    4. If sufficient storage for CESUBXP is not available, *
      *       a storage condition is generated by LE.             *
      *    5. CECNDHD get control and sets resume at the          *
      *       next instruction following the call to CESUBXP.     *
      *    6. A test for completion of CESUBXP is made after      *
      *       the subroutine call. If CESUBXP did not complete,   *
      *       a large amount of storage is freed, and CESUBXP     *
      *       is invoked a second time.                           *
      *    7. CESUBXP runs successfully once it has enough        *
      *       storage available.                                  *
      * Note: In order for this example to complete successfully, *
      *       the FREE suboption of the HEAP runtime option must  *
      *       be in effect.                                       *
      *************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    CECNDXP.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  TOKEN                   PIC X(4).
       01  HEAPID                  PIC S9(9) BINARY.
       01  HPSIZE                  PIC S9(9) BINARY.
       01  NEWSIZE                 PIC S9(9) BINARY.
       01  ADDRSS                  PIC S9(9) BINARY.
       01  PGMPTR USAGE IS         PROCEDURE-POINTER.
       01  FEEDBACK.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) BINARY.
       01  COMPLETED               PIC X.
           88  RAN                 VALUE 'Y'.
           88  NOTRUN              VALUE 'N'.
```

Figure 53. COBOL Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler (Part 1 of 2)

```
 PROCEDURE DIVISION.
 0001-BEGIN-PROCESSING.
************************************************************
**  Register user condition handler CECNDHD using CEEHDLR  **
************************************************************
     SET PGMPTR TO ENTRY  'CECNDHD'.
     MOVE 97 TO TOKEN
     CALL 'CEEHDLR' USING PGMPTR TOKEN.
     MOVE 0 TO HEAPID.
************************************************************
**   Allocate large amount of heap storage                **
************************************************************
     MOVE  500000 TO HPSIZE.
     CALL 'CEEGTST' USING HEAPID, HPSIZE, ADDRSS, FEEDBACK.
     IF CEE000 OF FEEDBACK THEN
************************************************************
**       Call CESUBXP, which requires a large stack        **
************************************************************
         SET NOTRUN TO TRUE
         CALL 'CESUBXP' USING COMPLETED
************************************************************
*        Check whether CESUBXP completed, or failed with    *
*        storage condition. If CESUBXP did not run,         *
*        resize the heap element down by a large amount     *
*        and call it again.                                 *
************************************************************
         IF NOTRUN THEN
             DISPLAY 'Reduce storage acquired BY main program'
                     ' AND CALL CESUBXP again.'
             MOVE 300 TO NEWSIZE
             CALL 'CEECZST' USING ADDRSS, NEWSIZE
             CALL 'CESUBXP' USING COMPLETED
         END-IF
       ELSE
         DISPLAY 'Call TO GET Storage Failed WITH MESSAGE '
             Msg-No OF FEEDBACK
     END-IF.

     GOBACK.
 END PROGRAM CECNDXP.
```

Figure 53. COBOL Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler (Part 2 of 2)

When any condition occurs in CECNDXP (Figure 53), CECNDHD (Figure 54) receives control and tests for the out-of-storage condition. If the out-of-storage condition has occurred, then CECNDHD calls CEEMRCR to return to the instruction in CECNDXP after the call to CESUBXP.

```
CBL LIB,APOST,NODYNAM
      *Module/File Name: IGZTOOSH
      ***********************************************************
      *                                                         *
      * CECNDHD - Call CEEMRCR to move the resume cursor        *
      *           relative to the handle cursor.                *
      *                                                         *
      * CECNDHD is a user condition handler that is registered  *
      * by the program CECNDXP.  CECNDHD gets control from the  *
      * condition manager and tests for the STORAGE CONDITION.  *
      * If a STORAGE CONDITION is detected, the resume cursor   *
      * is moved so that control is returned to the caller of   *
      * the routine encountering the STORAGE CONDITION.         *
      *                                                         *
      ***********************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CECNDHD.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  Movetyp                   PIC S9(9) BINARY.
       01  Feedback.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) BINARY.
       LINKAGE SECTION.
       01  Current-condition.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) BINARY.
      **
       01  Token                   PIC X(4).
      **
```

Figure 54. COBOL User-Written Condition Handler Registered for the Out-of-Storage Condition (Part 1 of 2)

```
 01  Result-code            PIC S9(9) BINARY.
     88  resume             VALUE +10.
     88  percolate          VALUE +20.
     88  perc-sf            VALUE +21.
     88  promote            VALUE +30.
     88  promote-sf         VALUE +31.
 01  New-condition.
     02  Condition-Token-Value.
     COPY  CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) BINARY.
             04  Msg-No      PIC S9(4) BINARY.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) BINARY.
             04  Cause-Code  PIC S9(4) BINARY.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info           PIC S9(9) BINARY.

 PROCEDURE DIVISION USING current-condition, token,
                          result-code, new-condition.
***********************************************************
**  Determine if entry was for OUT OF STORAGE condition. **
***********************************************************
     IF CEE0PD OF current-condition THEN
        DISPLAY 'COBOL subroutine could NOT RUN because',
                ' of the insufficient storage condition.'
***********************************************************
**      Call CEEMRCR to move the resume cursor          **
***********************************************************
        MOVE 0 TO Movetyp
        CALL 'CEEMRCR' USING Movetyp, Feedback
        IF CEE000 OF Feedback THEN
           SET resume TO TRUE
        ELSE
           SET promote TO TRUE
           MOVE feedback TO new-condition
        END-IF
     ELSE
        SET percolate TO TRUE
     END-IF

     GOBACK.
 END PROGRAM CECNDHD.
```

Figure 54. COBOL User-Written Condition Handler Registered for the Out-of-Storage Condition (Part 2 of 2)

Figure 55 is a COBOL subroutine that causes the out-of-storage condition.

```
CBL LIB,APOST,NODYNAM
      *Module/File Name: IGZTOOSS
      ***********************************************************
      *                                                         *
      *   CESUBXP -                                             *
      *                                                         *
      *   When CESUBXP gets control, a request is made to LE    *
      *   to allocate storage for the declared array W2. An     *
      *   out of storage condition takes place, provided the    *
      *   caller has not allocated a large amount of storage.   *
      *                                                         *
      ***********************************************************
       IDENTIFICATION DIVISION.

       PROGRAM-ID.    CESUBXP.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  ARRAY.
           05  W2                  PIC X OCCURS 3000000 TIMES.
       LINKAGE SECTION.
       01  PARM1                   PIC X.
           88  RAN-OK VALUE 'Y'.

       PROCEDURE DIVISION USING PARM1.
       PARA-CND01A.
           MOVE 'B' TO W2(2999999).
           SET RAN-OK TO TRUE.

           GOBACK.
       End program CESUBXP.
```

Figure 55. COBOL Subroutine that Causes Out-of-Storage Condition

Signaling and Handling a Condition in a C Routine

Figure 56 shows how a user-written condition handler gains control for a condition that was signaled using CEESGL, and calls CEEGQDT to access a data structure that was set up in the signaling routine. The CEEMRCR callable service resets the resume cursor, and execution resumes at the new point:

```
/*Module/File Name:  EDCSIGH */
/*********************************************************************/
/* This example shows how several of the LE/VSE Condition Management  */
/* Callable Services are used.  The services shown are:               */
/* CEEHDLR  -- register a user condition handler                      */
/* CEESGL   -- signal a condition to the condition manager            */
/* CEEGQDT  -- get the q_data_token                                   */
/* CEEMRCR  -- move the resume cursor                                 */
/*                                                                    */
/* The example also shows how to directly construct a condition token */
/* and provides a sample user condition handler.                      */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

void b(void);

void handler(_FEEDBACK *,_INT4 *,_INT4 *,_FEEDBACK *);

typedef struct {          /* condition info structure */
   int   error_value;
   char err_msg[80];
   int   retcode;
} info_struct;

int main(void) {

  printf("In main program\n");
   b();
  /* CEEMRCR should put the resume cursor at this point */
  printf("Finished\n");
}
void b(void) {

  _FEEDBACK fc,condtok;
  _ENTRY routine;
  _INT4 token,qdata;
  info_struct *info;
  _INT2 c_1,c_2,cond_case,sev,control;
  _CHAR3 facid;
  _INT4 isi;
```

Figure 56. Sample C Calls to CEEHDLR, CEESGL, CEEGQDT, and CEEMRCR (Part 1 of 3)

```
  printf("In routine b\n");
  token = 99;
  routine.address = (_POINTER)&handler;
  routine.nesting = NULL;
  /* register the condition handler: handler */
  CEEHDLR(&routine,&token,&fc);

  if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
     printf("CEEHDLR failed with message number %d\n",
            fc.tok_msgno);
     exit(2999);
  }

  /* build the condition token */
  c_1 = 1;
  c_2 = 99;
  cond_case = 1;
  sev = 1;
  control = 0;
  memcpy(facid,"ZZZ",3);
  isi = 0;

  CEENCOD(&c_1,&c_2,&cond_case,&sev,&control,
          facid,&isi,&condtok,&fc);
  if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
     printf("CEENCOD failed with message number %d\n",
            fc.tok_msgno);
     exit(2999);
  }

  /* set up the condition info structure */
  info = (info_struct *)malloc(sizeof(info_struct));
  if (info == NULL) {
     printf("error allocating info_struct\n");
     exit(2399);
  }
  memset(info->err_msg,' ',79);
  info->err_msg[79] = '\0';
  info->error_value = 86;
  memcpy(info->err_msg,"Test message",12);
  info->retcode = 99;

  /* set qdata to be the condition info structure */
  qdata = (int)info;
  /* signal the condition */
  CEESGL(&condtok,&qdata,NULL);

  printf("Failed: handler should have moved resume cursor past this\n");
}

/*********************************************************************/
/* User condition handler                                            */
/*********************************************************************/
void handler(_FEEDBACK *fc, _INT4 *token, _INT4 *result,
             _FEEDBACK *newfc) {

   _FEEDBACK cursorfc, orig_fc;
   _INT4 type;
   _INT4 qdata;

   /* if the condition is not mine (ZZZ facid) then percolate */
   if (memcmp(fc->tok_facid,"ZZZ",3) != 0) {
      *result = 20;
      return;
   }
```

Figure 56. Sample C Calls to CEEHDLR, CEESGL, CEEGQDT, and CEEMRCR (Part 2 of 3)

```
   printf("%d is handling the condition for Control\n",*token);

   /* get the q_data_token */
   CEEGQDT(fc,&qdata,NULL);

   /* look at the q_data_token and print out a message if the */
   /* error_value was 86                                      */
   if (((info_struct *)qdata)->error_value == 86)
      printf("%.80s\n",((info_struct*)qdata)->err_msg);

   /* move the resume cursor to the caller of the routine */
   /* that registered this handler                        */
   type = 1;
   CEEMRCR(&type,&cursorfc);
  if ( _FBCHECK ( cursorfc , CEE000 ) != 0 ) {
     printf("CEEMRCR failed with message number %d\n",
            cursorfc.tok_msgno);
     exit(2999);
  }

   /* mark the condition as handled and return */
   printf("Condition handled\n");
   *result = 10;
   return;
}
```

Figure 56. Sample C Calls to CEEHDLR, CEESGL, CEEGQDT, and CEEMRCR (Part 3 of 3)

Handling a Divide-by-Zero Condition in a COBOL Program

Figure 57 illustrates how a COBOL program can handle a divide-by-zero condition if one should occur. The following occurs:

1. The program uses CEEHDLR to register a user-written condition handler that recognizes the divide-by-zero condition.
2. The program then performs a divide-by-zero, which causes the user-written condition handler to get control.
3. The handler calls CEE5GRN (Get Name of Routine that Incurred Condition), to return the name of the routine that the condition occurred in.
4. The handler inserts the routine name and condition token into a user-defined message string, and calls CEEMOUT (Dispatch a Message) to send the message to the LE/VSE message file.

   (The LE/VSE message file is a file that you can specify to store messages from a given routine or application, or from all routines that run under LE/VSE.)

```
CBL LIB,APOST,NODYNAM
      *Module/File Name: IGZTSIGR
      *****************************************************************
      **                                                              *
      ** IGZTSIGR -  Call the following LE services:                  *
      **                                                              *
      **              : CEEHDLR - register user condition handler     *
      **              : CEE5GRN - get name of routine that incurred   *
      **                          condition.                          *
      **              : CEEMOUT - output message associated with      *
      **                          condition, including name of        *
      **                          routine that incurred condition.    *
      **                                                              *
      **  1. Our example registers user condition handler IGZTSIGH.   *
      **  2. Our program then divides by zero which causes a          *
      **     hardware exception condition.                            *
      **  3. IGZTSIGH gets control and prints out a message           *
      **     that includes the name of the routine incurring          *
      **     the divide-by-zero condition, IGZTSIGR.                  *
      **  4. IGZTSIGH requests that LE Condition management resume    *
      **     execution after the point at which the condition occurred *
      **  5. IGZTSIGR terminates normally.                            *
      **                                                              *
      *****************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID.    IGZTSIGR.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  DIVISOR         PIC S9(9) BINARY.
       01  QUOTIENT        PIC S9(9) BINARY.
      **
      **  Declares for condition handling
      **
       01   PGMPTR     USAGE IS PROCEDURE-POINTER.
       01   FBCODE.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) BINARY.
       77   TOKEN      PIC X(4).
```

Figure 57. COBOL Example of a Main Routine that Registers User-Written Condition Handler and Causes Divide-by-Zero Condition (Part 1 of 4)

```
       PROCEDURE DIVISION.
       0001-BEGIN-PROCESSING.
           DISPLAY '****************************************'.
           DISPLAY 'IGZTSIGR COBOL Example is now in motion.'.
           DISPLAY '****************************************'.
      ** **********************************************************
      **  Register user condition handler IGZTSIGH using CEEHDLR
      ** **********************************************************
           SET PGMPTR TO ENTRY 'IGZTSIGH'.
           MOVE 97 TO TOKEN.
           CALL 'CEEHDLR' USING PGMPTR, TOKEN, FBCODE.
           IF ( NOT CEE000 of FBCODE ) THEN
               DISPLAY 'Error ' Msg-No of FBCODE
                  ' registering condition handler IGZTSIGH' UPON CONSOLE
               STOP RUN
           END-IF.
      ** *************************************************************
      **  Divide by zero to cause a hardware exception condition.
      **  Condition handler IGZTSIGH gets control and invokes
      **  CEE5GRN to obtain the name of the routine in which
      **  condition was raised.  IGZTSIGH then prints a message
      **  using CEEMOUT and passing the name 'LEASMSIG'.
      **  Control returns and normal termination takes place.
      ** **************************************************************
           MOVE 0 TO DIVISOR.
           DIVIDE  5 BY DIVISOR GIVING QUOTIENT.
           DISPLAY '************************************'.
           DISPLAY 'IGZTSIGR COBOL Example is now ended.'.
           DISPLAY '************************************'.
           GOBACK.
       End program IGZTSIGR .

CBL LIB,APOST,NODYNAM
      *************************************************************
      ** IGZTSIGH -  Call the following LE services:
      **
      **             : CEE5GRN - Get name of routine that
      **             :           incurred a condition.
      **             : CEEMOUT - output a user message
      **
      **  This is the user condition handler registered
      **  by IGZTSIGR.  It calls CEE5GRN to retrieve the name of
      **  the routine incurring the divide-by-zero condition.
      **  It then calls CEEMOUT to output the message.
      *************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. IGZTSIGH.
```

Figure 57. COBOL Example of a Main Routine that Registers User-Written Condition Handler and Causes Divide-by-Zero Condition (Part 2 of 4)

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  msgstr.
    02  VarStr-length       PIC S9(4) BINARY.
    02  VarStr-text.
        03  VarStr-char         PIC X,
                    OCCURS 0 TO 256 TIMES
                    DEPENDING ON VarStr-length
                       OF msgstr.
01  Feedback.
    02  Condition-Token-Value.
    COPY  CEEIGZCT.
        03  Case-1-Condition-ID.
            04  Severity    PIC S9(4) BINARY.
            04  Msg-No      PIC S9(4) BINARY.
        03  Case-2-Condition-ID
                  REDEFINES Case-1-Condition-ID.
            04  Class-Code  PIC S9(4) BINARY.
            04  Cause-Code  PIC S9(4) BINARY.
        03  Case-Sev-Ctl    PIC X.
        03  Facility-ID     PIC XXX.
    02  I-S-Info            PIC S9(9) BINARY.
77  rtn-name        PIC X(80).
77  msgdest         PIC S9(9) BINARY.
77  string-pointer  PIC S9(4) BINARY.
*
LINKAGE SECTION.
01  Current-condition.
    02  Condition-Token-Value.
    COPY  CEEIGZCT.
        03  Case-1-Condition-ID.
            04  Severity    PIC S9(4) BINARY.
            04  Msg-No      PIC S9(4) BINARY.
        03  Case-2-Condition-ID
                  REDEFINES Case-1-Condition-ID.
            04  Class-Code  PIC S9(4) BINARY.
            04  Cause-Code  PIC S9(4) BINARY.
        03  Case-Sev-Ctl    PIC X.
        03  Facility-ID     PIC XXX.
    02  I-S-Info            PIC S9(9) BINARY.
**
01  Token           PIC X(4).
**
01  Result-code     PIC S9(9) BINARY.
    88 resume           VALUE +10.
    88 percolate        VALUE +20.
    88 perc-sf          VALUE +21.
    88 promote          VALUE +30.
    88 promote-sf       VALUE +31.
01  New-condition.
    02  Condition-Token-Value.
    COPY  CEEIGZCT.
        03  Case-1-Condition-ID.
            04  Severity        PIC S9(4) BINARY.
            04  Msg-No          PIC S9(4) BINARY.
        03  Case-2-Condition-ID
                  REDEFINES Case-1-Condition-ID.
            04  Class-Code      PIC S9(4) BINARY.
            04  Cause-Code      PIC S9(4) BINARY.
        03  Case-Sev-Ctl    PIC X.
        03  Facility-ID     PIC XXX.
    02  I-S-Info        PIC S9(9) BINARY.
```

Figure 57. COBOL Example of a Main Routine that Registers User-Written Condition Handler and Causes Divide-by-Zero Condition (Part 3 of 4)

```
 PROCEDURE DIVISION USING current-condition, token
                          result-code, new-condition.
************************************************************
*    Check to see whether this routine was entered due to a
*    divide-by-zero exception, or due to some other condition.
************************************************************
     IF CEE349 OF current-condition THEN
************************************************************
*        (A divide-by-zero condition has occurred)
************************************************************
         SET resume TO TRUE
************************************************************
**   Call CEE5GRN to retrieve the name of program         **
**     incurring the divide-by-zero exception.  Build     **
**     user message and include the name of the program.  **
************************************************************
         CALL 'CEE5GRN' USING rtn-name, feedback
         IF ( NOT CEE000 OF feedback ) THEN
             DISPLAY 'Error ' Msg-No OF feedback
                 ' in obtaining program name.' UPON CONSOLE
             MOVE feedback TO new-condition
             SET promote TO TRUE
         ELSE
             MOVE 1 TO string-pointer
             MOVE 255 TO VarStr-length OF msgstr
             STRING 'The example program ' rtn-name
                  ' incurred a divide-by-zero exception.'
                 DELIMITED BY '  ' INTO VarStr-text OF msgstr
                 POINTER string-pointer
             SUBTRACT 1 FROM string-pointer,
                 GIVING VarStr-length OF msgstr
             MOVE 2 TO msgdest
************************************************************
**          Call CEEMOUT to output the user message.
************************************************************
             CALL 'CEEMOUT' USING msgstr, msgdest, feedback
             IF ( NOT CEE000 OF feedback ) THEN
                 DISPLAY 'Error in writing message string.'
                 MOVE feedback TO new-condition
                 SET promote TO TRUE
             END-IF
         END-IF
     ELSE
************************************************************
*        (A condition other than divide-by-zero has occurred)
************************************************************
         SET percolate TO TRUE
     END-IF

     GOBACK.

 END PROGRAM IGZTSIGH.
```

Figure 57. COBOL Example of a Main Routine that Registers User-Written Condition Handler and Causes Divide-by-Zero Condition (Part 4 of 4)

Handling a Program Check in an Assembler Routine

The Figure 58 illustrates how an assembler routine can handle a program check if one should occur. The following occurs:

1. The routine registers a user-written condition handler, LEASMHD3, that responds to a program check by calling CEE5DMP to request a dump.
2. The routine then calls a subroutine, LEASMHD2, that generates a program check.
3. LE/VSE gives control to the user-written condition handler.

Note that a condition handler to which LE/VSE gives control does not have to be link-edited into the same phase as the routine in which the condition occurs; a condition handler can be dynamically loaded and can possibly dynamically load other phases also.

```
SMP1     TITLE 'Sample of main program that registers a handler'        00010000
*                                                                       00020000
*        Symbolic Register Definitions and Usage                        00030000
*                                                                       00040000
R1       EQU   1            Parameter list address, 0 if no parms      00130000
R10      EQU   10           Base register for executable code          00140000
R12      EQU   12           LE Common Anchor Area address              00150000
R13      EQU   13           Save Area/Dynamic Storage Area address     00160000
R14      EQU   14           Return point address                       00170000
R15      EQU   15           Entry point address                        00180000
*                                                                       00190000
*        Prologue                                                       00200000
*                                                                       00210000
CEEHDRA  CEEENTRY AUTO=DSASIZ,    Amount of main memory to obtain      00220000
               MAIN=YES,          This routine is a MAIN program       00230000
               PPA=PPA1,          Program Prolog Area for this routine 00240000
               BASE=R10           Base register for executable code    00250000
*                                      constants, and static variables 00260000
         USING CEECAA,R12         LE Common Anchor Area addressability 00270000
         USING CEEDSA,R13         Dynamic Storage Area addressability  00280000
*                                                                       00290000
*        Announce ourselves                                             00300000
*                                                                       00310000
         WTO   'CEEHDRA Says "HELLO"',ROUTCDE=2                         00320000
*                                                                       00330000
*        Register User Handler                                          00340000
*                                                                       00350000
         LA    R1,USRHDLPP        Get addr of proc-ptr to Handler rtn  00360000
         ST    R1,PARM1           Make it 1st parameter                00370000
         LA    R1,TOKEN           Get addr of 32-bit token             00380000
         ST    R1,PARM2           Make it 2nd parameter                00390000
         LA    R1,0               Omit address for Feedback Code:      00400000
*                                      If an error occurs while        00410000
*                                      registering the handler, LE will 00420000
*                                      signal the condition, rather    00430000
*                                      than passing it back to caller  00440000
         ST    R1,PARM3           Make it 3rd parameter                00450000
         LA    R1,HDLRPLST        Point to parameter list for CEEHDLR  00460000
         CALL  CEEHDLR            Invoke CEEHDLR callable service AWI  00470000
*                                                                       00480000
*        Call subroutine to cause an exception                          00490000
*                                                                       00500000
         CALL  LEASMHD2                                                 00510000
*                                                                       00520000
*        Un-Register User Handler                                       00530000
*                                                                       00540000
```

Figure 58. Assembler Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler (Part 1 of 6)

```
         LA    R1,USRHDLPP        Get addr of proc-ptr to Handler rtn   00550000
         ST    R1,HDLUPRM1        Make it 1st parameter                 00560000
         LA    R1,FEEDBACK        Address for Feedback Code             00570000
         ST    R1,HDLUPRM2        Make it 2nd parameter                 00580000
         LA    R1,HDLUPLST        Point to parameter list for CEEHDLU   00590000
         CALL  CEEHDLU            Invoke CEEHDLU callable service AWI   00600000
*                                                                       00610000
*        Bid fond farewell                                              00620000
*                                                                       00630000
         WTO   'CEEHDRA Says "GOOD-BYE"',ROUTCDE=2                      00640000
*                                                                       00650000
*        Epilogue                                                       00660000
*                                                                       00670000
         CEETERM RC=4,MODIFIER=1  Terminate program                     00680000
*                                                                       00690000
*        Program Constants and Local Static Variables                   00700000
*                                                                       00710000
USRHDLPP DC    V(LEASMHD3),A(0)   Procedure-pointer to Handler routine  00720000
*                                                                       00730000
         LTORG ,                  Place literal pool here               00740000
         SPACE 3                                                        00750000
PPA1     CEEPPA ,                 Program Prolog Area for this routine  00760000
         EJECT                                                          00770000
*                                                                       00780000
*        Map the Dynamic Storage  Area (DSA)                            00790000
*                                                                       00800000
         CEEDSA ,           Map standard CEE DSA prologue               00810000
*                                                                       00820000
*        Local Automatic (Dynamic) Storage..                            00830000
*                                                                       00840000
HDLRPLST DS    0F           Parameter List for CEEHDLR                  00850000
PARM1    DS    A            Address of User-written Handler             00860000
PARM2    DS    A            Address of 32-bit Token                     00870000
PARM3    DS    A            Address of Feedback Code cond token         00880000
*                                                                       00890000
HDLUPLST DS    0F           Parameter List for CEEHDLR                  00900000
HDLUPRM1 DS    A            Address of User-written Handler             00910000
HDLUPRM2 DS    A            Address of Feedback Code cond token         00920000
*                                                                       00930000
TOKEN    DS    F            32-bit Token:  fullword whose *value* will 00940000
*                                be passed to the user handler each     00950000
*                                time it is called.                     00960000
*                                                                       00970000
FEEDBACK DS    CL12         Feedback Code condition token               00980000
*                                                                       00990000
DSASIZ   EQU   *-CEEDSA     Length of DSA                               01000000
         EJECT                                                          01010000
*                                                                       01020000
*        Map the Common Anchor Area (CAA)                               01030000
*                                                                       01040000
         CEECAA                                                         01050000
         END   CEEHDRA                                                  01060000
HDR2     TITLE 'Sample of subprogram that forces a program check'       01070000
```

Figure 58. Assembler Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler (Part 2 of 6)

```
*                                                                        01080000
*        Symbolic Register Definitions and Usage                         01090000
*                                                                        01100000
R1       EQU   1            Parameter list address, 0 if no parms        01110000
R11      EQU   11           Base register for executable code            01120000
R12      EQU   12           LE Common Anchor Area address                01130000
R13      EQU   13           Save Area/Dynamic Storage Area address       01140000
R14      EQU   14           Return point address                         01150000
R15      EQU   15           Entry point address                          01160000
*                                                                        01170000
*        Prologue                                                        01180000
*                                                                        01190000
LEASMHD2 CEEENTRY AUTO=DSASIZ,    Amount of main memory to obtain       *01200000
               PPA=PPA2,          Program Prolog Area for this routine *01210000
               MAIN=NO,           This program is a Subroutine         *01220000
               NAB=YES,           YES because called by LE enabled rtn *01230000
               BASE=R11           Base register for executable code,    01240000
*                                      constants, and static variables  01250000
         USING CEECAA,R12         LE Common Anchor Area addressability  01260000
         USING CEEDSA,R13         Dynamic Storage Area addressability   01270000
*                                                                        01280000
*        Announce ourselves                                              01290000
*                                                                        01300000
         WTO   'LEASMHD2 Says "HELLO"',ROUTCDE=2                         01310000
*                                                                        01320000
*        Cause Data Exception (LE Condition 3207)                        01330000
*                                                                        01340000
         XC    A,A                Clear to Binary Zeros                  01350000
*                                      (not a valid packed number)       01360000
         AP    A,=P'7'            Cause Data exception                   01370000
*                                                                        01380000
*        Say good-bye                                                    01390000
*                                                                        01400000
         WTO   'LEASMHD2 Says "GOOD-BYE"',ROUTCDE=2                      01410000
*                                                                        01420000
*        Epilogue                                                        01430000
*                                                                        01440000
         CEETERM RC=0             Terminate program                      01450000
         SPACE 3                                                         01460000
*                                                                        01470000
*        Program Constants and Local Static Variables                    01480000
*                                                                        01490000
PPA2     CEEPPA ,                 Program Prolog Area for this routine  01500000
*                                                                        01510000
         LTORG ,                  Place literal pool here                01520000
         EJECT                                                           01530000
*                                                                        01540000
*        Map the Dynamic Storage  Area (CAA)                             01550000
*                                                                        01560000
         CEEDSA ,           Map standard CEE DSA prologue                01570000
*                                                                        01580000
*        Local Automatic (Dynamic) Storage..                             01590000
*                                                                        01600000
A        DS    PL2          Packed operand (uninitialized)               01610000
*                                                                        01620000
DSASIZ   EQU   *-CEEDSA     Length of DSA                                01630000
         EJECT                                                           01640000
*                                                                        01650000
*        Map the Common Anchor Area (CAA)                                01660000
*                                                                        01670000
         CEECAA                                                          01680000
         END   ,                  of LEASMHD2                            01690000
SMP3     TITLE 'User-written condition handler'                          01700000
```

Figure 58. Assembler Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler (Part 3 of 6)

```
*                                                                       01710000
*       Symbolic Register Definitions and Usage                         01720000
*                                                                       01730000
R1       EQU   1            Parameter list address (upon entry)         01740000
R2       EQU   2            Work register                               01750000
R3       EQU   3            Parameter list address (after CEEENTRY)     01760000
R4       EQU   4            Will point to Result Code Argument          01770000
R10      EQU   10           Will point to Condition Token Argument      01780000
R11      EQU   11           Base register for executable code           01790000
R12      EQU   12           LE Common Anchor Area address               01800000
R13      EQU   13           Save Area/Dynamic Storage Area address      01810000
R14      EQU   14           Return point address                        01820000
R15      EQU   15           Entry point address                         01830000*
                                                                        01840000
*       Prologue                                                        01850000
*                                                                       01860000
LEASMHD3 CEEENTRY AUTO=DSASIZ,    Amount of main memory to obtain      *01870000
               PPA=PPA3,          Program Prolog Area for this routine *01880000
               MAIN=NO,           This program is a Subroutine         *01890000
               NAB=YES,           YES because called in LE environment *01900000
               PARMREG=R3,        R1 value is saved here               *01910000
               BASE=R11           Base register for executable code,    01920000
*                                      constants, and static variables  01930000
         USING CEECAA,R12         LE Common Anchor Area addressability  01940000
         USING CEEDSA,R13         Dynamic Storage Area addressability   01950000
         USING UHDLARGS,R3        User Handler Args addressability      01960000
*                                                                       01970000
*       Locate Arguments                                                01980000
*                                                                       01990000
         L     R10,@CURCOND       Get address of Condition Token        02000000
         USING $CURCOND,R10       Condition Token addressability        02010000
         L     R4,@RESCODE        Get address of Result Code            02020000
         USING $RESCODE,R4        Result Code addressability            02030000
*                                                                       02040000
*       Announce ourselves                                              02050000
*                                                                       02060000
         WTO   'LEASMHD3 Says "HELLO"',ROUTCDE=2                        02070000
*                                                                       02080000
*       Process Condition                                               02090000
*                                                                       02100000
         CLC   CURCOND(8),CEE347  Was this handler entered due to the   02110000
*                                      condition it was created to      02120000
*                                      deal with (data exception) ?     02130000
         BE    BADPDATA           Yes -- go process it                  02140000
*                                 No..                                  02150000
         MVC   RESCODE,=A(PERCOLAT)    Indicate PERCOLATE action        02160000
         B     OUT                Return to LE/VSE Condition Manager    02170000
*                                                                       02180000
BADPDATA EQU   *                  Processing for data exception:        02190000
         MVC   RESCODE,=A(RESUME) Indicate RESUME action                02200000
*                                                                       02210000
*       Call CEE5DMP to Dump machine state                              02220000
*                                                                       02230000
         LA    R1,DUMPTITL        Get address of Dump Title             02240000
         ST    R1,PARM1           Make it first parameter               02250000
         LA    R1,DUMPOPTS        Get address of Dump Options string    02260000
         ST    R1,PARM2           Make it second parameter              02270000
         LA    R1,FC              Address of Feedback Code              02280000
         ST    R1,PARM3           Make it third parameter               02290000
         LA    R1,DMPPARMS        Point to parameter list for CEE5DMP   02300000
         CALL  CEE5DMP            Invoke CEE5DMP callable service AWI   02310000
```

Figure 58. Assembler Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler (Part 4 of 6)

```
*                                                                       02320000
*       Sign-off                                                        02330000
*                                                                       02340000
OUT      EQU   *                                                        02350000
         WTO   'LEASMHD3 Says "GOOD-BYE"',ROUTCDE=2                     02360000
*                                                                       02370000
*       Epilogue                                                        02380000
*                                                                       02390000
         CEETERM RC=0                                                   02400000
*                                                                       02410000
*       Program Constants and Local Static Variables                    02420000
*                                                                       02430000
DUMPOPTS DC    CL256'THR(ALL) BLOCK STORAGE'     Dump Options           02440000
*                                                                       02450000
DUMPTITL DC    CL80'LEASMHD3 - Sample Dump '     Dump Title             02460000
*                                                                       02470000
PPA3     CEEPPA ,                Program Prolog Area for this routine  02480000
*                                                                       02490000
         LTORG ,                 Place literal pool here                02500000
*                                                                       02510000
*       Define Symbolic Value Constants for Condition Tokens            02520000
*                                                                       02530000
         CEEBALCT                                                       02540000
         EJECT                                                          02550000
*                                                                       02560000
*       Map Arguments to User-Written Condition Handler                 02570000
*                                                                       02580000
UHDLARGS DSECT                                                          02590000
@CURCOND DS    A            Address of CIB                              02600000
@TOKEN   DS    A            Address of 32-bit token value from CEEHDLR 02610000
@RESCODE DS    A            Address of Result Code                      02620000
@NEWCOND DS    A            Address of New Condition                    02630000
         SPACE 3                                                        02640000
$CURCOND DSECT ,            Mapping of the current condition            02650000
CURCOND  DS    A            Condition token that identifies the         02660000
*                                current condition being processed      02670000
         SPACE 3                                                        02680000
$TOKEN   DSECT ,            Mapping of the 32-bit Token Argument        02690000
TOKEN    DS    A            Value of 32-bit Token from CEEHDLR call     02700000
         SPACE 3                                                        02710000
$RESCODE DSECT ,            Mapping of Result Code Argument             02720000
RESCODE  DS    F            Result Code specifies the action for        02730000
*                           the LE/VSE condition manager to take when  02740000
*                           control returns from the user handler:     02750000
RESUME   EQU   10           Resume at the resume cursor                 02760000
*                           (condition has been handled)               02770000
PERCOLAT EQU   20           Percolate to the next condition handler     02780000
*                           (if a Result Code is not explicitly set     02790000
*                            by the handler, this is the default)       02800000
PROMOTE  EQU   30           Promote to the next condition handler       02810000
*                           (New Condition has been set)                02820000
* (See the LE/VSE Programming Guide for other Result Code values)       02830000
         SPACE 3                                                        02840000
$NEWCOND DSECT ,            Mapping of the New Condition Argument       02850000
NEWCOND  DS    CL12         New Condition (condition token) specifies  02860000
*                                the condition promoted to.             02870000
         EJECT                                                          02880000
```

Figure 58. Assembler Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler (Part 5 of 6)

```
*                                                                      02890000
*       Map the Dynamic Storage  Area (DSA)                            02900000
*                                                                      02910000
        CEEDSA ,           Map standard CEE DSA prologue               02920000
*                                                                      02930000
*       Local Automatic (Dynamic) Storage..                            02940000
*                                                                      02950000
DMPPARMS DS 0F             Parameter list for CEE5DMP                  02960000
PARM1    DS    A           Address of Title string                     02970000
PARM2    DS    A           Address of Options string                   02980000
PARM3    DS    A           Address of Feedback Code                    02990000
*                                                                      03000000
FC       DS    CL12        Feedback Code condition token               03010000
*                                                                      03020000
DSASIZ   EQU   *-CEEDSA    Length of DSA                               03030000
         EJECT                                                         03040000
*                                                                      03050000
*       Map the Common Anchor Area (CAA)                               03060000
*                                                                      03070000
         CEECAA                                                        03080000
         END   ,                of LEASMHD3                            03090000
```

Figure 58. Assembler Example of a Main Routine that Calls Subroutine and Registers User-Written Condition Handler (Part 6 of 6)

Chapter 14. Using Condition Tokens

LE/VSE uses the 12-byte condition token data type to perform a variety of communication functions. This chapter describes the format of the condition token and its components, and how you can use the condition token to react to conditions and communicate conditions with other routines.

Understanding the Basics

If you provide an *fc* parameter in a call to an LE/VSE callable service, the service sets *fc* to a specific value called a condition token and returns it to your application. (See “The Effect of Coding the fc Parameter” on page 173 for more information.)

If you do not specify the *fc* parameter in a call to an LE/VSE service, LE/VSE generates a condition token for any nonzero condition and signals it using the CEESGL callable service. [4] Signaling the condition token causes it to be passed it to LE/VSE condition handling. (See “The Effect of Omitting the fc Parameter” on page 175 for more information.)

The condition token is used by the routines of your application to communicate with message services, the condition manager, and other routines within the application. For example, you can use it with LE/VSE message services to write a diagnostic message associated with a particular condition to a file. You can also determine if a particular condition has occurred by testing the condition token, or a symbolic representation of it. See “User-Written Condition Handler Interface using CEEHDLR” on page 138 for more information on coding user-written condition handlers. The structure of the condition token is discussed in “Understanding the Structure of the Condition Token” on page 172, and symbolic feedback codes are discussed in “Using Symbolic Feedback Codes” on page 175.

Related Services

LE/VSE provides callable services to help you construct and decompose your own condition tokens.

CEEDCOD Breaks down a condition token into its component parts.

CEENCOD Creates a new condition token in your application.

See *LE/VSE Programming Reference* for a detailed explanation of each field in a condition token and for more information on using CEEDCOD and CEENCOD callable services. See also the message handling services listed in Chapter 15, “Using and Handling Messages,” on page 185.

4. COBOL programs must provide the feedback code parameter in each call to an LE/VSE callable service. However, C and PL/I routines do not have to. See *LE/VSE Programming Reference* for information on how to provide the feedback code parameter in each HLL.

© Copyright IBM Corp. 1991, 2005

Understanding the Structure of the Condition Token

Figure 59 illustrates the structure of the condition token, with bit offsets shown above the components:

0 - 31	32 - 33	34 - 36	37 - 39	40 - 63	64 - 95
Condition_ID	Case Number	Severity Number	Control Code	Facility_ID	ISI token

For Case 1 condition tokens, Condition_ID is:

0 - 15	16 - 31
Severity Number	Message Number

For Case 2 condition tokens, Condition_ID is:

0 - 15	16 - 31
Class Code	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.

Figure 59. LE/VSE Condition Token

Every condition token contains the components indicated in Figure 59:

Condition_ID A 4-byte identifier that, with the facility ID, describes the condition that the token communicates. The format of Condition_ID depends on whether a Case 1 (service condition) or Case 2 (class/cause code) condition is being represented. LE/VSE callable services and most applications can produce Case 1 conditions. Case 2 conditions may be produced by some operating systems and compiler libraries. LE/VSE does not produce them directly.

Figure 59 illustrates the format of the Condition_ID for Case 1 and Case 2 conditions.

Case Specifies whether the condition token is for a Case 1 or Case 2 condition.

Severity Specifies the severity of the condition represented by the condition token.

Control Specifies whether the facility ID has been assigned by IBM.

Facility ID A 3-character alphanumeric string that identifies the product or component of a product that generated the condition; in the case of LE/VSE, the facility ID is CEE. Although all LE/VSE-conforming HLLs use LE/VSE message and condition handling services, the actual run-time messages generated under LE/VSE still carry the language identification in the facility ID. The facility ID for PL/I, for example, is IBM.

When paired with a message number, a facility ID uniquely identifies a message in the message source file. The facility ID and message number persist throughout an application. This allows the

meaning of the condition and its associated message to be determined at any point in the application after a condition has occurred.

If you are creating a new facility ID to use with your own message source file, follow the guidelines listed under the *Facility_ID* parameter of CEENCOD in *LE/VSE Programming Reference*.

If you create a new facility_ID to use with a message source file to be processed by CEEBLDTX ("Creating Messages" on page 185), be aware that the facility ID must be part of the message source file name. It is therefore important to follow the naming guidelines in order to have a module name that does not abend.

ISI
A 4-byte instance specific information (ISI) token associated with a given instance of the condition. A nonzero ISI token indicates the presence of instance specific information. The ISI can contain data on message inserts for the message associated with the condition. It can also contain a 4-byte token (q_data_token) that represents qualifying data (q_data) that user-written condition handlers use to identify and react to a specific condition. The q_data_token can contain q_data, or it can be a pointer to q_data.. The ISI is typically built by LE/VSE for system or LE/VSE-signaled conditions. It can also be built by an application for conditions signaled using CEESGL. The CEECMI callable service can be used to define the message inserts within the ISI for a condition token. The q_data to be placed in the ISI for a condition token is defined when signaling the condition using CEESGL.

A user-written condition handler can retrieve information from the ISI. Message insert information cannot be retrieved directly; however, the entire formatted message with inserts can be formatted and placed in an application-provided character string using CEEMGET. The q_data_token can be retrieved using CEEGQDT.

The Effect of Coding the fc Parameter

The feedback code is the last parameter of all LE/VSE callable services, and the second to last parameter of all LE/VSE math services. [4] When the *fc* parameter is provided and a condition is raised, the following sequence of events occurs:

1. The callable service in which the condition occurred builds a condition token for the condition. The condition token is a 12-byte representation of an LE/VSE condition. Each condition is associated with a single LE/VSE run-time message.
2. The callable service places information into the ISI, which might contain the following:
   - A timestamp.
   - Information that is inserted into a message associated with the condition.

     For example, you can use the CEEBLDTX utility (see "Creating Messages" on page 185) or the CEECMI callable service (see *LE/VSE Programming Reference*) to generate message inserts. Routines signaling a new condition with a call to CEESGL should first call CEECMI to copy any insert information into the ISI associated with the condition.
3. If the severity of the detected condition is critical (severity = 4), it is raised directly to the condition manager. LE/VSE then processes the condition in the manner described in "Condition Step" on page 109.

4. If the condition severity is not critical (severity less than 4), the condition token is returned to the routine that called the service.
5. When the condition token is returned to your application, you can use the condition token in the following ways:
   - Ignore it and continue processing.
   - Signal it to LE/VSE using the CEESGL callable service.
   - Get, format, and dispatch the message for display using the CEEMSG callable service.
   - Store the message in a storage area using the CEEMGET callable service.
   - Use the CEEMOUT callable service to dispatch a user-defined message string to a destination that you specify.
   - Compare the condition token to one that is known to you so that you can react appropriately. You can test the condition token for success, equivalence or equality.

   See *LE/VSE Programming Reference* for more information about LE/VSE callable services.

Testing a Condition Token for Success

To test a condition token for success, it is sufficient to determine if the first 4 bytes are zero; if the first 4 bytes are zero, the remainder of the condition token is zero, indicating that a successful call was made to the service.

The LE/VSE condition handling model provides two ways you can check for success using the *fc* parameter. You can compare the value returned in *fc* to the symbolic feedback code CEE000, or you can compare it to a 12-byte condition token containing all zeroes coded in your routine. See "Using Symbolic Feedback Codes" on page 175 for details.

You do not necessarily need to check the feedback code after every invocation of a service or to check for success before proceeding with execution. However, if you want to ensure that your application is invoking callable services successfully, test the feedback code after each call to a service.

Testing Condition Tokens for Equivalence

Two condition tokens are equivalent if they represent the same type of condition, even if not necessarily the same instance of the condition. For example, you could have two occurrences of an out-of-storage condition. Though equivalent conditions, they are not necessarily equal because they occur in different locations in your program.

To determine whether two condition tokens are equivalent, compare the first 8 bytes of each condition token to one another. These bytes are static and do not change depending upon the given instance of the condition.

Two reasons you might check for equivalence are to write a message about a type of condition that occurs in your application or to register a condition handling routine to respond to a given type of condition.

There are two ways to check for equivalent condition tokens:

- You can break down the condition token by coding it as a structure and looking at its individual components, or you can call the CEEDCOD (decompose condition token) service to break down the condition token. See *LE/VSE Programming Reference* for more information about the CEEDCOD service.

- The easiest way to test for equivalence is to compare the value returned in *fc* with the symbolic feedback code for the condition you are interested in handling. Symbolic feedback codes represent only the first 8 bytes of a 12-byte condition token. See "Using Symbolic Feedback Codes" for details.

Testing Condition Tokens for Equality

To determine whether two condition tokens are equal (that is, the same instance or occurrence of the condition token), you must compare all 12 bytes of each condition token with each other. The last 4 bytes can change from instance to instance of a given condition.

The only way to test condition tokens for equality is to compare the value returned in *fc* with another condition token that has either been returned from a call to a service, or that you have coded as a 12-byte condition token in your routine. Symbolic feedback codes are used to test for equivalence; they are not useful in testing for equality because they represent only the first 8 bytes of the condition token.

The Effect of Omitting the fc Parameter

When a feedback code is not provided, any nonzero condition is raised. Signaled conditions are processed by LE/VSE in the manner described in "Condition Step" on page 109. If the condition remains unhandled at the end of processing, LE/VSE takes the LE/VSE default action (defined in Table 28 on page 111). The message delivered is the translation of the condition token into English (or another supported national language).

Using Symbolic Feedback Codes

LE/VSE provides symbolic feedback codes representing the first 8 bytes of a 12-byte condition token. Using LE/VSE-provided symbolic feedback codes saves you from having to define an 8-byte condition token in your code whenever you want to check for the occurrence of a condition. Symbolic feedback codes are limited to testing for conditions rather than actual condition instances: no ISI information is tested using symbolic feedback codes because the comparison is only performed against the first 8 bytes of the condition token.

LE/VSE provides include files (copy files) that define all LE/VSE symbolic feedback codes. See "Including Symbolic Feedback Code Files" on page 176 for information about LE/VSE symbolic feedback code files.

Locating Symbolic Feedback Codes for Conditions

In LE/VSE you can locate symbolic feedback codes in the following ways:

- Look in the first column of the symbolic feedback codes table listed after each of the callable services in *LE/VSE Programming Reference*. The symbolic feedback code table for the CEEGTST (get heap storage) callable service is shown in Table 34.

Table 34. Symbolic Feedback Codes Associated with CEEGTST

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.

Table 34. Symbolic Feedback Codes Associated with CEEGTST (continued)

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE0P2	4	0802	Heap storage control information was damaged.
CEE0P3	3	0803	The heap identifier in a get storage request or a discard heap request was unrecognized.
CEE0P8	3	0808	Storage size in a get storage request (CEEGTST) or a reallocate request (CEECZST) was not a positive number.
CEE0PD	3	0813	Insufficient storage was available to satisfy a get storage (CEECZST) request.

If you want to test for the condition raised when you specify an invalid heap ID from which to get storage, you can compare the symbolic feedback code CEE0P3 to the condition token returned either from the service or from the LE/VSE condition manager (depending on whether you specified *fc* in the call to CEEGTST).

- If you want to code a condition handling routine to handle a condition resulting in an error message from your application, look up the error message in *LE/VSE Debugging Guide and Run-Time Messages*. You will find the symbolic feedback code for the condition listed there.

Including Symbolic Feedback Code Files

Symbolic feedback codes are provided for LE/VSE, C, COBOL, and PL/I conditions.

If you installed LE/VSE in the default sublibraries, the symbolic feedback code files are stored in the PRD2.SCEEBASE sublibrary. To use symbolic feedback codes, you must include the symbolic feedback code files in your source code.

The symbolic feedback code files have file names of the form *xxxyyy*CT, where:

xxx
: Indicates the facility ID of the conditions represented in the file. For example, EDC*yyy*CT contains condition tokens for C-specific conditions (those with the facility ID of EDC).

  xxx can be CEE (LE/VSE), EDC (C), IBM (PL/I), or IGZ (COBOL).

yyy
: Indicates the facility ID of the language in which the declarations are coded. For example, EDCIBMCT contains PL/I declarations of C condition tokens. *yyy* can be BAL (assembler), EDC (C), IBM (PL/I), or IGZ (COBOL).

CT
: Stands for “condition token.”

To use symbolic feedback codes, include the file in your source code using the appropriate language construct. For example:

- In C, specify:

  ```
  #include <ceeedcct>
  #include <igzedcct>
  ```

  to include the files containing the C declarations for CEE (LE/VSE) and IGZ (COBOL) condition tokens.
- In COBOL, specify:

  ```
         COPY CEEIGZCT.
         COPY IGZIGZCT.
  ```

  to include the files containing the COBOL definitions of CEE (LE/VSE) and IGZ (COBOL) condition tokens.
- In PL/I, specify:

  ```
   %INCLUDE CEEIBMCT;
   %INCLUDE EDCIBMCT;
  ```

  to include the files containing the PL/I definitions of CEE (LE/VSE) and EDC (C) condition tokens.

Before compiling your source program, you need to include the sublibrary containing the include (copy) files in the source library search chain, as shown below:

```
// LIBDEF SOURCE,SEARCH=(PRD2.SCEEBASE)
```

Examples Using Symbolic Feedback Codes

The following examples use symbolic feedback codes to test user input and display a message if the input is incorrect.

C

In the following example, the symbolic feedback code file CEEEDCCT is included and a call is made to CEEGTST. After the call, a test is made for the condition token representing an invalid heap ID. The *fc* returned from CEEGTST is tested against the symbolic feedback code CEE0P3 listed in the CEEGTST feedback code table (see *LE/VSE Programming Reference*). If the heap ID specified is invalid, another call is made to CEEGTST to try again.

_FBCHECK (IBM-supplied) is used to compare only the first 8 bytes of the *fc* against the symbolic feedback code.

```
/*Module/File Name:  EDCSFC   */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>


main(void)
{
  _FEEDBACK fc;
  _POINTER address;
  _INT4 heapid, size;

  size = 1000;
  heapid = 999;

  CEEGTST(&heapid, &size, &address, &fc);
  if ((_FBCHECK (fc, CEE0P3)) == 0){
     printf("You specified a Heap Id that doesn't exist!\n\n");
     }
  printf("Try again:\n");
  heapid = 0;

  CEEGTST(&heapid, &size, &address, &fc);
  if ((_FBCHECK (fc, CEE000)) == 0){
     printf("Now it worked!\n");
     }
  else {
     printf("CEEGTST failed with message number%d \n", fc.tok_msgno);
     exit(99);
     }


  return 0;
}
```

Figure 60. C Example Testing for CEEGTST Symbolic Feedback Code CEE0P3

COBOL

In the following example, the symbolic feedback code file CEEIGZCT is accessed and a call is made to CEESDEXP (exponential base e). The first 8 bytes of the feedback code returned are tested against the symbolic feedback code CEE1UR to ensure that the input parameter is within the valid range for CEESDEXP. The symbolic feedback code table for CEESDEXP is listed in *LE/VSE Programming Reference*. A message is displayed if the input parameter is out of range.

```
CBL LIB,APOST
      * Module/File Name: IGZTSFC
      *************************************************
      *                                               *
      * CTDEMO -  This routine assigns values to a    *
      *           condition token.                    *
      *                                               *
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CTDEMO.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 FBC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) BINARY.
       01 X COMP-2 VALUE +2.0E+02.
       01 Y COMP-2.
       PROCEDURE DIVISION.
           CALL 'CEESDEXP' USING X FBC Y.
           IF CEE1UR of FBC THEN
               DISPLAY 'Argument X out of range'
                       ' for CEEDEXP'
           END-IF

           GOBACK.
```

Figure 61. COBOL Example Testing for CEESDEXP Symbolic Feedback Code CEE1UR

It is important that symbolic feedback codes be compared with only the first 8 bytes of the 12-byte condition token. To this end, you must code the COPY statements for the symbolic feedback code declarations in the right place within the condition token declaration.

In Figure 61, for example, symbolic feedback code `CEE1UR` is compared to the first 8 bytes of condition token `FBC` because of the correct placement of the COPY statements.

It is wrong to place the COPY statements before the declaration of `Condition-Token-Value` as shown in Figure 62 on page 180, because the 8-byte symbolic feedback code blank-padded (`X'40'`) to a length of 12 bytes would be compared to the full 12-byte condition token. The comparison would fail, because the blanks would not match the ISI data in the last 4 bytes of the condition token.

```
 01 FBC
     COPY CEEIGZCT.                   <-----+  Incorrect
     COPY IGZIGZCT.                   <-----+  Incorrect
     02  Condition-Token-Value
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) BINARY.
             04  Msg-No      PIC S9(4) BINARY.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) BINARY.
             04  Cause-Code  PIC S9(4) BINARY.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) BINARY.
⋮
```

Figure 62. Wrong Placement of COBOL COPY Statements for Testing Feedback Code

PL/I

The following example includes the symbolic feedback code file CEEIBMCT so that LE/VSE feedback codes (those with facility ID CEE) will be defined. FBCHECK (IBM-supplied) is called to compare the first 8 bytes of FC with the symbolic feedback code CEE000 to determine if the call to CEEMGET is successful. If it is, the message associated with feedback code CEE001 is printed.

```
*PROCESS MACRO;
 /*Module/File Name: IBMMGET                          **/
 /****************************************************/
 /**                                                ***/
 /**Function       : CEEMGET - Get a Message        **/
 /**                                                ***/
 /****************************************************/
 PLIMGET: PROC OPTIONS(MAIN);

   %INCLUDE  CEEIBMAW;
   %INCLUDE  CEEIBMCT;

   DCL 01 CONTOK   FEEDBACK;
   DCL 01 FC       FEEDBACK;
   DCL MSGBUF      CHAR(80);
   DCL MSGPOINTER  INT4;

   /* Give CONTOK value of condition CEE001         */
   ADDR( CONTOK ) -> CEEIBMCT = CEE001;
   MSGPTR = 0;

   /* Call CEEMGET to retrieve msg corresponding    */
   /*    to condition token                         */
   CALL CEEMGET ( CONTOK, MSGBUF, MSGPTR, FC );
   IF  FBCHECK( FC, CEE000)  THEN  DO;
      PUT SKIP LIST( 'Message text for message number'
         || CONTOK.MsgNo || ' is "' || MSGBUF || '"');
      END;
   ELSE  DO;
      DISPLAY( 'CEEMGET failed with msg '
         || FC.MsgNo );
      STOP;
      END;
```

Figure 63. PL/I Example Testing for Symbolic Feedback Code CEE000

Condition Tokens for C Signals under C

You need the condition token representing an event as input to many LE/VSE condition and message handling services. C signals have condition token representations that you can use for this purpose. The signals listed in Table 35 have a condition token representation with facility ID of EDC.

Table 35. LE/VSE Condition Tokens and C Signals

Sev-erity	Message Number	Symbolic Feedback Code	Case	Sev-erity	Control	ID	Signal Name	Signal Number
3	6000	EDC5RG	1	3	1	EDC	SIGFPE	8
3	6001	EDC5RH	1	3	1	EDC	SIGILL	4
3	6002	EDC5RI	1	3	1	EDC	SIGSEGV	11
3	6003	EDC5RJ	1	3	1	EDC	SIGABND	18
3	6004	EDC5RK	1	3	1	EDC	SIGTERM	15
3	6005	EDC5RL	1	3	1	EDC	SIGINT	2
2	6006	EDC5RM	1	2	1	EDC	SIGABRT	3
3	6007	EDC5RN	1	3	1	EDC	SIGUSR1	16
3	6008	EDC5RO	1	3	1	EDC	SIGUSR2	17
1	6009	EDC5RP	1	1	1	EDC	SIGIOERR	27

LE/VSE-provided q_data Structure for Abends

When LE/VSE intercepts an abend or VSE cancel, one of the following LE/VSE messages is issued and the corresponding condition raised:

Message Number	Condition
CEE3250C	CEE35I
CEE3321C	CEE37P
CEE3322C	CEE37Q

For all abends and VSE cancels except VSE cancel code 20 (program check), LE/VSE provides q_data as part of the ISI token for the condition. The q_data can be retrieved using the CEEGQDT callable service from within CEEHDLR-established condition handlers. See *LE/VSE Programming Reference* for syntax of the CEEGQDT service. The q_data associated with abends is also listed by message number in the *LE/VSE Debugging Guide and Run-Time Messages*.

Qualifying data is comprised of a list of addresses pointing to information that can be used by HLL and user-written condition handlers to react to a condition. The q_data structure is shown in Figure 64 on page 183.

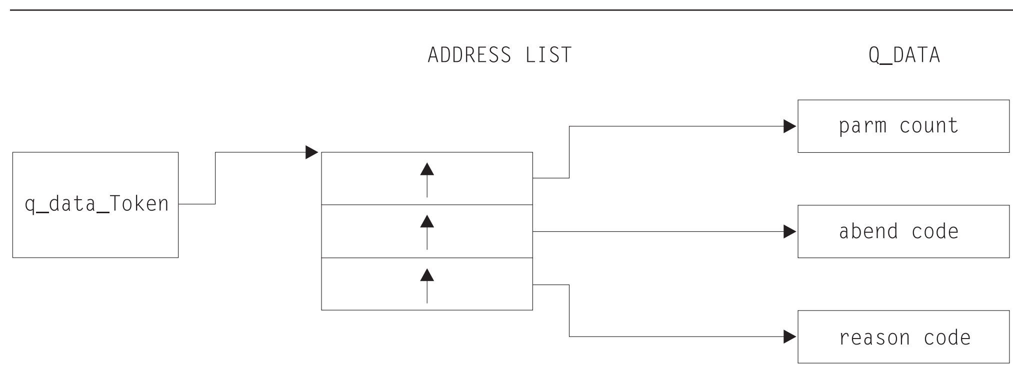


Figure 64. Structure of Abend Qualifying Data

The information provided by the q_data_token is input only and is described below.

parm count (input)
A fullword field containing the total number of meaningful parameters in the q_data structure, including the *parm count*. In this case, the value of *parm count* is a fullword containing the integer 3.

abend code
A 4-byte field containing the abend code or the VSE cancel code. The VSE cancel code is what is contained in the low-order byte of R0 on entry to the STXIT routine.

reason code
A 4-byte field containing the reason code. If a reason code is not available (as occurs, for example, in a CICS abend), *reason code* is set to zero.

Chapter 15. Using and Handling Messages

This chapter describes how you can use the LE/VSE message services to create, issue, and handle messages for LE/VSE-conforming applications.

Understanding the Basics

The LE/VSE message services provide a common method of handling and issuing messages for LE/VSE-conforming applications.

When a condition is raised in your application, either LE/VSE common routines or language-specific run-time routines can issue messages from the run-time message file. The messages can provide information about the condition and suggest possible solutions to errors.

You can use LE/VSE callable services and run-time options to modify message handling, and control the destination of message output. You can also define a message log file to create a record of the messages that LE/VSE issues.

Related Options and Services

Run-Time Options

MSGFILE	Specifies a file where run-time messages issued by LE/VSE are logged
MSGQ	Specifies the maximum number of ISIs
NATLANG	Specifies the run-time national language

Callable Services

CEEMGET	Gets a message
CEEMOUT	Dispatches a message
CEEMSG	Gets, formats, and dispatches a message
CEECMI	Stores and loads message insert data about a condition

Utilities

CEEBLDTX	Builds a loadable message module

See *LE/VSE Programming Reference* for syntax information on run-time options and callable services.

Creating Messages

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

© Copyright IBM Corp. 1991, 2005

Creating a Message Source File

The message source file contains the message text and information associated with each message. Standard tags and format are used to describe the message text and message information. The tags and format of the message source files are used by the CEEBLDTX utility to transform the source file into an assembler-language file. This file is assembled, and the resultant object module is link-edited to produce a phase containing the loadable message module.

The source file should have a fixed record format with a record length of 80. When creating a message source file, make sure your sequential numbering attribute is turned off in the editor so that trailing sequence numbers are not generated. Trailing blanks in columns 1–72 are ignored.

One message source file is required for each national language.

All tags used to create the source file begin with a colon(:), followed by a keyword and a period(.). All tags must begin in column 1, except where noted. Comments in the message source file must begin with a period asterisk (.*) in the leftmost position of the input line.

Figure 65 shows an example of a message source file for messages with a facility ID of XMP.

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

Figure 65. Example of a Message Source File

The tags used in message source files are described below. Unless otherwise indicated, each tag is optional.

:facid. The facility ID is required at the beginning of every message file. It is used as the first 3 characters of the message number and the first 3 characters of the condition token. All messages within a source file have the same facility ID. For example, all messages issued by LE/VSE have a facility ID of CEE. The facility ID is combined with a 4-digit identification number and the message severity code to form the message number. The facility ID can contain any alphanumeric (A–Z, a–z, 0–9) characters. If this tag is omitted, or if this tag is specified more than once, the CEEBLDTX utility generates error messages.

To avoid confusion with message files used by IBM products, you should not use facility IDs CEE, EDC, EQA, IBM, IGZ, and FOR.

This tag is required.

:msgno. The message number tag defines the beginning and end of information for a message. All information up to the next `:msgno.`

tag refers to the current message. The message number appears as the 4 digits following the message prefix, and is used to identify the message in a message source file. Multiple messages can use the same message number, but only if a `:msgsubid.` tag is used within the message.

The message numbers used with the `:msgno.` tags must be in ascending order. The message numbers can be from 1 to 4 numeric (0–9) characters. Leading zeros will be added if fewer than 4 digits are specified.

This tag is required.

:msgsubid.
The message subidentifier tag distinguishes between different messages with the same message number. If every message has a unique message number, the `:msgsubid.` tag is unnecessary.

The numbers specified with the `:msgsubid.` tags must be unique and in ascending order within messages that have the same message number. The numbers can be from 1 to 4 numeric (0–9) characters. Leading zeros will be added if fewer than 4 digits are used.

:msgname.
The `:msgname.` tag is used to give a name to a message. This name becomes the symbolic name (feedback code) of the condition token associated with the message, and is placed into the copy member generated by the CEEBLDTX utility. For example, if EXMPLMSG is used for the `:msgname.` tag in a message, the symbolic feedback code for the condition associated with this message is `EXMPLMSG`.

If a message name is omitted, the facility ID plus the base 32 equivalent of the message number is used as the symbolic message name. For example, if the `:msgno.` is 10 and the facility ID is XMP, the symbolic feedback code for the condition associated with this message is `XMP00A`.

If the `:msgsubid.` tag is used, the subidentifier preceded by an underscore is added to the message name. For example, if the `:msgno.` is 10, the `:msgsubid.` is 1, and the facility ID is XMP, the symbolic feedback code for the condition associated with this message is `XMP00A_0001`.

:msgclass.
The `:msgclass.` (or `:msgcl.`) tag makes up the final part of the message identification. It requires a case-sensitive character that indicates the severity code of the message. This character corresponds to the level of severity of the condition token associated with the message. If the `:msgclass.` tag differs from the severity level of the condition token, the severity assigned to the condition token is used. Refer to Table 36 on page 198 for the severity codes, levels of severity, and condition descriptions.

This tag is required.

:msg.
The `:msg.` tag indicates the beginning of partial or complete text of the message to be displayed. The message text can appear in any national language known to LE/VSE (including DBCS characters). For a list of the supported national languages, refer to *LE/VSE Programming Reference*. The `:msg.` tag can be repeated as often as necessary to construct a message. It is not required if the message consists only of message inserts. If the message text for a message

requires more than one line, all lines are left-aligned with the beginning of the first line of message text.

The message text ends with the last nonblank character. There is no fixed space reserved for the message, so there is no requirement to reserve any additional space for message translation.

:hex. The `:hex.` tag indicates the beginning of a hexadecimal character string. If used, it must be within the text of a `:msg.` tag. It is terminated by an `:ehex.` tag. The `:hex.` tag can occur anywhere within the message text.

:ehex. The `:ehex.` tag terminates a string of hexadecimal characters. This tag can occur anywhere within the message text.

:dbc. The `:dbc.` tag defines text of DBCS characters. The string itself cannot contain any SBCS characters, and it must begin with a shift-out character and end with a shift-in character.

:tab.n The `:tab.` tag indicates that the next part of the message will be tabbed over a given number of spaces or tabbed to a given column. If the number is preceded by a plus sign, it indicates the next part of the message will be moved over the specified number of spaces from the current position. Otherwise, the number indicates the column where the next message part will begin. The tab value must be between 1 and 255. If necessary, a new line of output is automatically created in order to accommodate the tab value. This includes the case where the current position is greater than a specified tab column.

:tbn. The `:tbn.` tag is used to force any text written on a subsequent line to start in the current column until an `:etbn.` tag is found.

:etbn. The `:etbn.` tag turns off the tabs set by a `:tbn.` tag.

:ins n.[text] The `:ins.` tag defines a message insert. The insert is a variable that is assigned a value with the CEECMI callable service. The insert number (*n*) can be any number between 1 and 9. The text following the period describes the insert. This text is optional, and is included only in a message file when the value assigned to the insert is not known. For example, the text *variable name* after an insert tag indicates that a variable name is assigned to the insert.

One value can be assigned to each insert used in a message. Insert tags can be moved around, interchanged, or omitted, but the insert values cannot be changed. The order of the `:ins n.` tags, not the insert number, determines the order of the inserts.

:newline. The `:newline.` tag creates a new message line that can be used for multi-line messages.

:xpl. The `:xpl.` tag indicates text used to explain the condition. It is not printed as part of the message, but is included if the message source file is formatted and printed.

:presp. The `:presp.` tag indicates text that describes the suggested programmer response. It is not printed as part of the message, but is included if the message source file is formatted and printed.

:sysact. The `:sysact.` tag indicates text that describes the system action. It

is not printed as part of the message, but is included if the message source file is formatted and printed.

Using the CEEBLDTX Utility

CEEBLDTX is a utility that transforms the message source file into an assembler-language file that can be assembled and link-edited into a phase containing the loadable message module. The CEEBLDTX utility also optionally produces a language-specific copy book that contains declarations for the condition tokens associated with each message.

You can run the CEEBLDTX utility in the following environments:

- VSE batch

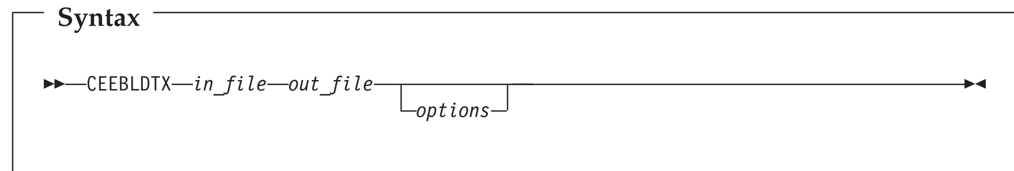
  To run the CEEBLDTX utility on VSE, you must have the REXX/VSE Interpreter installed on your VSE system. For more information about the REXX/VSE Interpreter, see *REXX/VSE User's Guide*

- CMS

  To run the utility on CMS, you must first copy the CEEBLDTX source program, CEEBLDTX.PROC, from the PRD2.SCEEBASE sublibrary to a CMS-accessible minidisk as CEEBLDTX EXEC.

The syntax of the CEEBLDTX invocation is shown below.

Syntax



in_file

- On VSE, either SYSIPT or a fully-qualified librarian member name in the format:

  in_lib.in_sublib.in_member.in_type

- On CMS, the file name of the SCRIPT file containing the message text source.

out_file

- On VSE, the fully-qualified name of the librarian member containing the assembler-language text version of the messages. The fully-qualified member name has the format:

  out_lib.out_sublib.out_member.out_type

  If *in_file* is SYSIPT, *out_type* must not be SCRIPT.

- On CMS, the filename of the resulting ASSEMBLE file containing the text version of the messages.

options

Can be omitted or one of the following:

COBOL(*copy-member-name*)
PLI(*copy-member-name*)
BAL(*copy-member-name*)

where *copy-member-name*

- On VSE, is the fully-qualified name of the librarian member containing the copy member. The fully-qualified member name has the format:

`copy_lib.copy_sublib.copy_member.copy_type`

If you plan to include the copy member generated by the CEEBLDTX utility in your application program, you should specify *copy_type* according to the specified language:

C for COBOL
P for PL/I
A for BAL

If *in_file* is SYSIPT, *copy_type* must not be SCRIPT.

- On CMS, is the CMS filename with the following default file types based on the specified language:

  COBOL for COBOL
  PLIOPT for PL/I
  MACRO for BAL

Usage Notes:

1. Each parameter is positional. Every parameter, except the *options* parameter, is required.
2. On CMS, an equal sign (=) can be substituted for any parameter, except for *in_file*. Parameters represented by an equal sign (=) are equated with the corresponding parameter previously used.

Files Created by CEEBLDTX

When you run the CEEBLDTX utility on VSE, the utility creates up to three librarian members:

out_lib.*out_sublib*.*out_member*.**SCRIPT**
: This librarian member is only created if *in_file* is specified as SYSIPT, and contains a copy of the source file read from SYSIPT.

out_lib.*out_sublib*.*out_member*.*out_type*
: The librarian member containing the assembler-language version of the message file. The source statements in this member can be assembled and link-edited into a phase containing the loadable message module. When the name of the loadable message file is placed in a message module table, the LE/VSE message services can dynamically access the file. See "Creating a Message Module Table" on page 195 for more information about creating a message module table.

copy_lib.*copy_sublib*.*copy_member*.*copy_type*
: The librarian member containing the language-specific COPY or INCLUDE member. This member contains the declarations for the condition tokens associated with each message in the message file. When this member is included in the source routine, the condition tokens can be used to reference the message. The `:msgname.` tag indicates the symbolic name of the condition token.

When you run the CEEBLDTX utility on CMS, the utility creates one or two files:

out_file **ASSEMBLE**
: The ASSEMBLE file can be assembled into a loadable text file. When the name of this file is placed in a message module table, the LE/VSE message services can dynamically access the file. See "Creating a Message Module Table" on page 195 for more information about creating a message module table.

copy_member_name **COPY or** *copy_member_name* **INCLUDE**
: The COPY or INCLUDE file contains the declarations for the condition

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

Running the CEEBLDTX Utility

Once you have created a message source file, you need to run the CEEBLDTX utility to create the assembler-language version of the file.

Running the CEEBLDTX Utility on VSE

The following sample JCL shows how you can invoke the REXX/VSE Interpreter to run the CEEBLDTX utility. The message source file being processed is in the librarian member EXAMPLE.SCRIPT in the sublibrary USER.SUBLIB. The sample JCL shows the parameters needed to produce the assembler-language version of the message file in USER.SUBLIB sublibrary member EXMPLASM.A, and the COBOL copy member in the USER.SUBLIB sublibrary member EXMPLCOB.C. The label information for the USER library is assumed to be in system standard labels.

```
Column 1       Column 16                                          Column 72
↓              ↓                                                  ↓
// JOB CEEBLDTX
// LIBDEF PROC,SEARCH=PRD2.SCEEBASE
// EXEC   REXX=CEEBLDTX,PARM='USER.SUBLIB.EXAMPLE.SCRIPT              X
               USER.SUBLIB.EXMPLASM.A COBOL(USER.SUBLIB.EXMPLCOB.C)'
/&
```

The following sample JCL shows how to invoke the REXX/VSE Interpreter on VSE to run the CEEBLDTX utility to process a message source file provided in SYSIPT. The sample JCL shows the parameters needed to produce the assembler-language version of the message file in USER.SUBLIB sublibrary member EXMPLASM.A, and the PL/I include member in the USER.SUBLIB sublibrary member EXMPLPLI.P. The label information for the USER library is assumed to be in system standard labels.

```
Column 1       Column 16                                          Column 72
↓              ↓                                                  ↓
// JOB CEEBLDTX
// LIBDEF PROC,SEARCH=PRD2.SCEEBASE
// EXEC   REXX=CEEBLDTX,PARM='SYSIPT USER.SUBLIB.EXMPLASM.A           X
               PLI(USER.SUBLIB.EXMPLPLI.P)'
⋮
message source file
/*
/&
```

The following sample JCL shows an alternative way to invoke the REXX/VSE Interpreter on VSE to run the CEEBLDTX utility to process a message source file provided in SYSIPT and produce the output described in the previous example.

```
Column 1       Column 16                                          Column 72
↓              ↓                                                  ↓
// JOB CEEBLDTX
// LIBDEF PROC,SEARCH=PRD2.SCEEBASE
// EXEC   ARXJCL,PARM='CEEBLDTX SYSIPT USER.SUBLIB.EXMPLASM.A         X
               PLI(USER.SUBLIB.EXMPLPLI.P)'
⋮
message source file
/*
/&
```

Running the CEEBLDTX Utility on CMS

To use the CEEBLDTX utility with the sample file shown in Figure 65 on page 186 you would issue:

```
CEEBLDTX example exmplasm pli(exmplpli)
```

The *in_file* is EXAMPLE SCRIPT, the *out_file* is EXMPLASM ASSEMBLE, and the PL/I include file is EXMPLPLI PLIOPT.

Assembling and Link-Editing the Message File

To assemble and link-edit the assembler-language file (EXMPLASM) produced by the CEEBLDTX utility, use the following JCL.

```
// JOB    EXMPLASM
// LIBDEF SOURCE,SEARCH=(USER.SUBLIB,PRD2.SCEEBASE)
// LIBDEF PHASE,CATALOG=USER.SUBLIB
// OPTION CATAL
   PHASE  EXMPLASM,*
// EXEC   ASMA90,SIZE=ASMA90
   COPY   EXMPLASM
/*
// EXEC   LNKEDT
/&
```

CEEBLDTX Error Messages

LE/VSE issues the CEEBLDTX error message described below. In some situations, a CEEBLDTX message might be accompanied by a REXX Interpreter message which can help you determine the cause of the problem.

When running the CEEBLDTX utility on VSE, one or more REXX/VSE Interpreter messages might be produced without an accompanying CEEBLDTX error message. These messages usually indicates a recoverable error. You can ignore the following REXX/VSE Interpreter messages if they occur **without** an accompanying CEEBLDTX error message:

```
ARX0563E Unable to open file member_name

ARX0670E EXECIO error while trying to GET or PUT a record.
```

Return Code=0028 Message source file ***ssssssss*** **not found.**

Explanation: On VSE, the librarian member with the fully-qualified name *ssssssss* could not be found.

On CMS, the file with the name *ssssssss* could not be found an any accessed disk.

Programmer Response: On VSE, make sure the name of the message source file is specified correctly as SYSIPT or a fully-qualified librarian member name. If you specify a librarian member name, make sure that you provide label information for the specified library.

ON CMS, make sure the file is correctly named and the file is available on one of your accessed disks.

Return Code=0036 Disk A must be accessed as Read/Write.

Explanation: On CMS, the A-disk must be writeable to write the output files.

Programmer Response: Specify an A-disk that is write accessible.

Return Code=0040 Error on line ***nnn*** **in message** ***nnnn*** **Insert number greater than mmmm.**

Explanation: An insert number greater than the allowable maximum was specified. The current maximum allowable insert number is 9.

Programmer Response: Specify an insert number of 9 or less.

Return Code=0044 Error on line ***nnn*** **Duplicate :FACID. tags found within the given message source file.**

Explanation: Only one facility ID can be specified in the message source file.

Programmer Response: Specify only one facility ID in the message source file.

Return Code=0048 No :FACID. tag found within the given message source file.

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

Programmer Response: Specify a 3-character facility ID with the :facid. tag.

Return Code=0052 Error on line *nnn* Message number *nnnn* found out of range mmmm to mmmm.

Explanation: A message was found with a number outside the valid range. The current valid range is 0 to 9999.

Programmer Response: Correct the invalid message number on the given line of the message source file.

Return Code=0056 Number of hex digits not divisible by 2 on line *nnn* in message *nnnn*.

Explanation: A hexadecimal string must contain an even number of digits.

Programmer Response: Specify an even number of digits for the hexadecimal string.

Return Code=0060 Invalid hexadecimal digits on line *nnn* in message *nnnn*.

Explanation: Valid hexadecimal digits are 0–9 and A–F. Invalid digits were detected.

Programmer Response: Specify only digits 0–9 and A–F within a hexadecimal string.

Return Code=0064 Number of DBCS bytes not divisible by 2 on line *nnn* in message *nnnn*.

Explanation: Double-byte character strings must contain an even number of bytes.

Programmer Response: Specify an even number of bytes for the double-byte character string.

Return Code=0068 ASSEMBLE out_file name must be longer than the message facid *pppp*.

Explanation: The ASSEMBLE file name must be greater than 3 characters.

Programmer Response: Specify an ASSEMBLE out_file name of greater than 3 characters.

Return Code=0072 Message facility ID *pppp* on line *nnn* was longer than 4 characters.

Explanation: Facility ID must be exactly 3 characters long, with no blanks.

Programmer Response: Specify a 3-character facility ID.

Return Code=0076 Message class on line *nnn* was not a valid message class type: IWESCFLA.

Explanation: Message class must be one of the valid message classes.

Programmer Response: Specify a valid message class.

Return Code=0080 Error on line *nnn* - tag not recognized.

Explanation: A tag that was not recognized was encountered.

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

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

Explanation: The first tag of the message source file must be the facility ID tag.

Programmer Response: Specify the facility ID tag as the first tag in the message source file.

Return Code=0088 Error on line *nnn* - unexpected tag.

Explanation: A valid tag was found in an unexpected location in the message source file. It is likely out of order.

Programmer Response: Check the order of the tags in the message source file.

Return Code=0092 Error on line *nnn* - duplicate tags *ttt*.

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

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

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

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

Programmer Response: Specify at least one message in the message file.

Return Code=0100 Error on line *nnn* - insert number was not provided or was less than 1.

Explanation: A positive insert number must be provided for each insert.

Programmer Response: Specify a positive insert number of 9 or less for the insert.

Return Code=0104 Error on line *nnn* in message *mmmm* - subid *ssss* found out of range 0 to 9999.

Explanation: A message subid was found with a number outside the valid range. The current valid range is 0 to 9999.

Programmer Response: Correct the invalid message subid on the given line of the message source file.

Return Code=0108 Existing *copy_member_name* COPY file found, but not on A-disk.

Explanation: On CMS, a feedback token file was found with the given name, but it is not on the A-disk, and will not be replaced.

Programmer Response: Specify a different feedback token file name, or release the disk on which the file currently resides.

Return Code=0120 Error *rc* opening file *filename* for *io_type* processing.

Explanation: The REXX I/O command, EXECIO, returned the return code *rc* when trying to open file *filename* for *io_type* (READ or WRITE) processing.

Programmer Response: See your *REXX Reference* for a description of the EXECIO return code, and correct the problem that caused the return code.

Return Code=0124 Error *rc* copying file *in_file* to *out_file*.

Explanation: The REXX I/O command, EXECIO, returned the return code *rc* when writing to the file *out_file*.

Programmer Response: See your *REXX Reference* for a description of the EXECIO return code, and correct the problem that caused the return code.

Return Code=0128 Invalid option *language*(*copy_member_name*).

Explanation: The language specified for the optional copy member was not COBOL, PLI, or BAL.

Programmer Response: Specify COBOL, PLI, or BAL for the language type of the copy member.

Return Code=0132 Variable used on line *lineno* has not been initialized.

Explanation: The REXX NOVALUE condition has been signaled. One of the REXX variables used on line *lineno* of the CEEBLDTX source file has not been initialized. This error should not occur.

Programmer Response: If you have modified the CEEBLDTX source file, correct the problem by initializing any uninitialized REXX variables.

If you have not modified the CEEBLDTX source file, this is an internal problem. Contact your service representative.

Return Code=*nnn* Undefined error number *nnn* issued.

Explanation: An undefined error was encountered.

Programmer Response: Contact your service representative.

Creating a Message Module Table

LE/VSE locates the user-created messages using a message module table that you code in assembler.

The message module table begins with a header that indicates the number of languages in the table. In Figure 66, for example, only English is used, so the first fullword of the header declares the constant F'1'.

```
        TITLE 'UXMPMSGT'
UXMPMSGT CSECT
         DC   F'1'              number of languages
         DC   CL8'ENU     '     language identifier
         DC   A(TABLEENU)       pointer to first language table
TABLEENU DC   F'01'             lowest message number in module
         DC   F'100'            highest message number in module
         DC   CL8'EXMPLASM'     message module name
         DC   F'-1'             flags indicating the last...
         DC   F'-1'               16-byte entry (a dummy entry)...
         DC   CL8'DUMMY'          in the language table
         END  UXMPMSGT
```

Figure 66. Example of a Message Module Table with One Language

In Figure 67, however, both English and Japanese are used, so the first fullword of the header declares the constant F'2'. Following the message module table header are tables for each language.

```
         TITLE 'UZOGMSGT'
UZOGMSGT CSECT
         DC   F'2'              number of languages
         DC   CL8'ENU     '     first language identifier
         DC   A(TABLEENU)       pointer to first language table
         DC   CL8'JPN     '     second language identifier
         DC   A(TABLEJPN)       pointer to second language table
TABLEENU DC   F'01'             lowest message number in first module
         DC   F'100'            highest message number in first module
         DC   CL8'ZOGMSGE1'     first message module name
         DC   F'101'            lowest message number in second module
         DC   F'200'            highest message number in second module
         DC   CL8'ZOGMSGE2'     second message module name
⋮
         DC   F'-1'             flags indicating the last...
         DC   F'-1'               16-byte entry (a dummy entry)...
         DC   CL8'DUMMY'          in the language table
TABLEJPN DC   F'01'             lowest message number in first module
         DC   F'100'            highest message number in first module
         DC   CL8'ZOGMSGJ1'     first message module name
         DC   F'101'            lowest message number in second module
         DC   F'200'            highest message number in second module
         DC   CL8'ZOGMSGJ2'     second message module name
⋮
         DC   F'-1'             flags indicating the last...
         DC   F'-1'               16-byte entry (a dummy entry)...
         DC   CL8'DUMMY'          in the language table
         END  UZOGMSGT
```

Figure 67. Example of a Message Module Table with Two Languages

Each language table has one or more 16-byte entries that indicate the name of a phase containing a loadable message module, and the range of message numbers the module contains. The first fullword of each 16-byte entry contains the lowest

message number within the corresponding module; the second fullword contains the highest message number for that module. The last 8 bytes of each 16-byte entry contain the phase name of the loadable message module. For example, in Figure 67 on page 195, Japanese messages numbered 101–200 are found in phase ZOGMSGJ2. Finally, each language table ends with a dummy 16-byte entry whose first two fullwords contain the flag `F'-1'` indicating the end of the language table.

Use an 8-character format for the title of the message module table: 'U' (to indicate that the table contains user-created messages), followed by a 3-character facility ID, followed by 'MSGT'. For example, the title of the message module table for messages using a facility ID of XMP would be 'UXMPMSGT' as shown in Figure 66 on page 195; the title of the message module table for messages having a facility ID of ZOG would be 'UZOGMSGT' as shown in Figure 67 on page 195.

After you create the message module table:

1. Assemble it and link-edit it into a loadable phase
2. Store the phase in a sublibrary where it can be dynamically accessed while your application is running

The following shows an example of how to assemble and link-edit a message module table.

```
// JOB     UXMPMSGT
// DLBL    IJSYSLN,'%LEVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,       X
                 RECORDS=(400,600)
// LIBDEF  PHASE,CATALOG=(USER.RUNLIB)
// OPTION  CATAL
   PHASE   UXMPMSGT,*
// EXEC    ASMA90,SIZE=ASMA90
⋮
message module table source
⋮
/*
// EXEC    LNKEDT
/*
/&
```

Figure 68. Assembling and Link-Editing a Message Phase Table

Assigning Values to Message Inserts

After you add message insert tags to the message source file, you can use the LE/VSE callable service CEECMI to assign values to the inserts. Values do not need to be assigned to inserts in sequential order. For example, the value of insert 3 can be assigned before the value for insert 1. Before invoking the CEECMI callable service, assign values to the callable service parameters. For more information about CEECMI, see *LE/VSE Programming Reference*.

Figure 69 on page 197 shows an example of the use of CEECMI to assign value 1234 to insert 1 for `:msgname.EXMPLMSG` shown in Figure 65 on page 186.

```
*PROCESS MACRO;
 TEST: Proc Options(Main);

 /*Module/File Name: IBMMINS          */

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    %INCLUDE SYSLIB(EXMPLCOP);
    DCL INSERT    CHAR(255) VARYING  AUTO;
    DCL 01 CTOK   FEEDBACK;
    DCL 01 FBCODE FEEDBACK;
    DCL MSGFILE   FIXED AUTO;

    ctok = EXMPLMSG;
    insert = '1234';
    MSGFILE = 2;

    /* Call CEECMI to create a message insert */
    CALL CEECMI(ctok, 1, insert, fbcode);

    /* Call CEEMSG to issue the message */
    CALL CEEMSG(ctok, MSGFILE, fbcode);

 END TEST;
```

Figure 69. Example of Assigning Values to Message Inserts

Using Messages in Code

Figure 70 shows a test case in which the object module for program TEST (Figure 69) is link-edited and run. The MSGFILE run-time option is used to direct the output to a file named TSTMSGS.

```
// JOB     TEST
// DLBL    IJSYSLN,'%LEVSE.WORKFILE.IJSYSLN',0,VSAM,RECSIZE=322,       X
               RECORDS=(400,600)
// DLBL    TSTMSGS,'TEST.MESSAGES.FILE',0,VSAM,RECSIZE=133,            X
               RECORDS=(100,10)
// LIBDEF  OBJ,SEARCH=(userlib.sublib,PRD2.SCEEBASE)
// LIBDEF  PHASE,SEARCH=(userlib.sublib,PRD2.SCEEBASE)
// OPTION  LINK
   INCLUDE TEST
// EXEC    LNKEDT
// EXEC    ,PARM='MSGFILE(TSTMSGS)/'
/*
/&
```

Figure 70. Example of Link-Editing and Running the TEST PL/I Program

The following shows the message output from the test case.

Example of Message Output

```
XMP0010I This is an example of an insert, 1234, within a message.
```

Interpreting Run-Time Messages

Run-time messages are designed to provide information about conditions and possible solutions to errors that occur in your routine. LE/VSE common routines and language-specific run-time routines issue run-time messages. All run-time messages in LE/VSE are comprised of the following:

- A 3-character facility ID used by all messages generated under LE/VSE or a particular LE/VSE-conforming product. This prefix indicates the LE/VSE component that generated the message, and is also the facility ID in the condition token. LE/VSE uses the ID of the condition token to write the message associated with the condition to MSGFILE. For more information about the condition token, see Chapter 14, “Using Condition Tokens,” on page 171.
- A message number that identifies the message associated with the condition.
- A severity level that indicates the severity of the condition that was raised.

The format of every run-time message is:

FFF*nnnnx*

FFF
: represents the facility ID. In LE/VSE Version 1 Release 4, the possible facility IDs assigned by IBM are:

CEE LE/VSE common library
EDC C language-specific library
IGZ COBOL language-specific library
IBM PL/I language-specific library

nnnn
: represents the message number.

x represents the severity code. This character indicates the level of severity (0, 1, 2, 3, or 4) of the message.

Table 36 lists the severity codes, corresponding severity levels, explanations of the severity codes, and the default actions taken if conditions corresponding to each level of severity are unhandled.

Table 36. LE/VSE Run-Time Message Severity Codes

Severity Code	Level of Severity	Explanation	Default Action If Condition Unhandled
I	0	An informational message (or, if the entire token is zero, no information)	No message issued.
W	1	A warning message. Service completed, probably successfully.	No message issued, except in COBOL. Processing continues for all languages.
E	2	Error detected, correction attempted, service completed, perhaps successfully.	Issues message and terminates thread.
S	3	Severe error detected, service incomplete with possible side effects.	Issues message and terminates thread.
C	4	Critical error detected, service incomplete with condition signaled.	Issues message and terminates thread.

LE/VSE messages can appear even though you made no explicit calls to LE/VSE services. C, COBOL, and PL/I run-time library routines commonly use the LE/VSE services, so you may receive LE/VSE messages even when the application routine does not directly call LE/VSE services.

Some LE/VSE conditions have qualifying data associated with the instance specific information (ISI) for the condition. For more information about qualifying data, refer to "LE/VSE-provided q_data Structure for Abends" on page 182.

Specifying National Language

You can use LE/VSE national language support to view run-time messages in uppercase or mixed-case U.S. English, or Japanese. You can also use national language support to select the most appropriate language variables for your messages, such as language character set, left-to-right text, single-byte character set (SBCS), and double-byte character set (DBCS).

LE/VSE message services support requirements for national language support machine-readable information such as message formatting, message delivery, and normalization (removes adjacent shift-out, shift-in character in order to make DBCS strings as compatible as possible).

The NATLANG run-time option allows you to set the national language used for messages before you run your routine. The default national language is uppercase U.S. English. Refer to *LE/VSE Programming Reference* for more information on the NATLANG run-time option.

The CEE5LNG callable service allows you to set or query the current national language setting while your routine is running. Refer to *LE/VSE Programming Reference* for more information about CEE5LNG.

Handling Message Output

The following sections provide information about directing message output and displaying messages under LE/VSE, C, COBOL, and PL/I.

For information about handling message output in ILC applications, see *LE/VSE Writing Interlanguage Communication Applications*.

Using LE/VSE MSGFILE

Run-time messages are directed to a common LE/VSE message file. You can use the MSGFILE run-time option to specify the filename of this file. If a message file filename is not declared, messages are written to the IBM-supplied default, SYSLST.

Table 37 lists the default attributes of the message file.

Table 37. Message File Default Attributes

MSGFILE	Default Attributes
Default (SYSLST)	Logical record length 133[1] Unblocked ASA print control character
Printer (SYS*nnn*)	Logical record length 133 Unblocked ASA print control character

Table 37. Message File Default Attributes (continued)

MSGFILE	Default Attributes
Unlabeled tape (SYS*nnn*)	Logical record length 133 Block size 133*100 ASA print control character
Disk file	Logical record length 133 Block size (133*100)+8 ASA print control character
Labeled tape file (SYS*nnn*)	Logical record length 133 Block size 133*100 ASA print control character

Note:

1. If SYSLST is assigned to a disk file on CKD DASD, the logical record length is 121.

When you direct run-time messages to an I/O device, the method you should use depends on the device type. Table 38 lists methods for directing run-time messages to each allowable type of I/O device.

Table 38. Defining an I/O Device for the Message File

MSGFILE Device Type	JCL to Define I/O Device
Default (SYSLST)	`// ASSGN SYSLST,`*cuu*
Printer (SYS*nnn*)	`// ASSGN SYS`*nnn*`,`*cuu*
Unlabeled tape (SYS*nnn*)	`// ASSGN SYS`*nnn*`,`*cuu*
Disk file (*filename*)	`// DLBL` *filename*...
Labeled tape file (SYS*nnn*)	`// TLBL  SYS`*nnn*`,... // ASSGN SYS`*nnn*`,`*cuu*

Notes:

1. You might need to modify existing JCL of pre-LE/VSE-conforming applications in order to define new filenames for MSGFILE.
2. You can specify the same message file across nested enclaves; LE/VSE coordinates the use of the same filename across nested enclaves. If you specify different MSGFILE filenames in each enclave, LE/VSE honors each filename.
3. Under CICS, the MSGFILE run-time option defaults to the CESE transient data queue. You can specify an alternative transient data queue for the MSGFILE run-time option. For more information about message handling and run-time message output under CICS, see "Run-Time Output under CICS" on page 308.

Using C Input/Output Functions

C makes a distinction between types of error output, and whether the output is directed to the MSGFILE destination or to one of the standard stream output devices, `stderr` or `stdout`.

Run-time messages and `perror()` messages are directed to the `stderr` standard stream output device. The default destination for `stderr` output is the MSGFILE filename; you can change this default as discussed below.

Message output issued by a call to the `printf()` function is directed to `stdout`. The default destination of `stdout` is SYSLST.

You can change the destination of `printf()` output by redirection. For example, `1>&2` in the PARM parameter of the JCL EXEC statement at routine invocation redirects `stdout` to the `stderr` destination.

Table 39 lists the types of C output, the types of messages associated with them, and the destination of the message output:

Table 39. C Message Output

Type of Output	Type of Message	Produced By	Default Destination
MSGFILE output	LE/VSE messages (CEExxxx)	LE/VSE unhandled conditions	MSGFILE filename
C library messages	C unhandled conditions	MSGFILE filename	
`stderr` messages	`perror()` messages (EDCxxx)	Issued by a call to `perror()`	MSGFILE filename
User output sent explicitly to `stderr`	Issued by a call to `fprintf()`	MSGFILE filename	
`stdout` messages	User output sent explicitly to `stdout`	Issued by a call to `printf()`	`stdout`

You can control the destination of `stderr` output by using the LE/VSE MSGFILE run-time option. You can control the destination of `stdout` output by using the C `freopen()` function or by requesting redirection services at run time.

Table 40 lists the possible destinations of redirected `stderr` and `stdout` standard stream output.

Table 40. C Redirected Stream Output

	stderr Not Redirected	stderr Redirected to Destination Other Than stdout	stderr Redirected to stdout
stdout not redirected	`stdout` to itself	`stdout` to itself	Both to `stdout`
`stderr` to MSGFILE	`stderr` to its other destination		
stdout redirected to destination other than stderr	`stdout` to its other destination	`stdout` to its other destination	Both to the other `stdout` destination
`stderr` to MSGFILE	`stderr` to its other destination		
stdout redirected to stderr	Both to MSGFILE	Both to the other `stderr` destination	When `stderr` and `stdout` are redirected to each other (this is not recommended), output from both is directed to whichever was specified first.

For more information about redirecting standard streams in C, see *LE/VSE C Run-Time Programming Guide*.

Using COBOL Input/Output Statements

LE/VSE directs run-time messages produced by VS COBOL II and COBOL/VSE routines to the file specified by the LE/VSE MSGFILE run-time option. If your application includes a DOS/VS COBOL program, run-time messages, such as those produced by the DOS/VS COBOL SYMDMP, STATE, FLOW, and COUNT compiler options, are directed to SYSLST.

LE/VSE manages all user-specified output directed to the system-logical output device. This includes output produced by the following statements:
- DISPLAY [UPON SYSLST]
- EXHIBIT (DOS/VS COBOL only)
- READY TRACE (DOS/VS COBOL only)

Note: The COBOL DISPLAY statement is not supported under CICS. The DOS/VS COBOL READY TRACE and EXHIBIT statements are also not supported under CICS.

LE/VSE determines the destination of user-specified output as follows:
- For a DOS/VS COBOL program in your application, the DISPLAY, READY TRACE, and EXHIBIT statements send all message output to SYSLST. This output will be synchronized with other run-time message output that is directed to SYSLST by the VS COBOL II and COBOL/VSE OUTDD compiler option.
- For a VS COBOL II program in your application, the DISPLAY statement sends message output to the file specified by the VS COBOL II OUTDD compiler option when the routine was compiled.

  Note: The VS COBOL II compiler under VSE does not support the OUTDD compiler option, and the system-logical output device is always SYSOUT. However, LE/VSE treats the OUTDD filename of SYSOUT as SYSLST. Therefore, if the VS COBOL II routine was compiled using the VS COBOL II compiler under VSE, output from the DISPLAY statement is sent to SYSLST.
- For a COBOL/VSE program in your application, the DISPLAY statement sends message output to the file specified by the COBOL/VSE OUTDD compiler option. For compatibility with COBOL compilers that run on z/OS and VM, the IBM-supplied default value for the OUTDD compiler option is SYSOUT. However, LE/VSE treats the OUTDD filename of SYSOUT as SYSLST.

  If the filename specified in the OUTDD compiler option matches the filename specified in the MSGFILE run-time option, the output is synchronized with the run-time messages.

If the file designated by the OUTDD compiler option or the MSGFILE run-time option has not been defined (associated with an I/O device) when the output is delivered, the LE/VSE condition with symbolic feedback code CEE0E9 is raised.

The possible filename specification combinations for OUTDD and MSGFILE and the locations where display output and run-time messages are routed are summarized in Table 41.

Table 41. Run-time Message and DISPLAY Destinations for OUTDD and MSGFILE filename Specifications

filename Specification	filename Defined?	Destination[1]
MSGFILE(SYSLST) OUTDD(SYSLST)	Yes, for SYSLST	Messages and DISPLAY data are routed to SYSLST.
	No	The LE/VSE condition with symbolic feedback code CEE0E9 is raised.

Table 41. Run-time Message and DISPLAY Destinations for OUTDD and MSGFILE filename Specifications (continued)

filename Specification	filename Defined?	Destination[1]
MSGFILE(SYSLST) OUTDD(filename)	Yes, for SYSLST	Messages are routed to SYSLST.
	Yes, for filename	COBOL/VSE DISPLAY data is routed to the file defined for filename.
	No	If the file designated by filename is not defined when output is directed to it, the LE/VSE condition with symbolic feedback code CEE0E9 is raised. If SYSLST is not defined when output is directed to it, the LE/VSE condition with symbolic feedback code CEE0E9 is raised.
MSGFILE(filename) OUTDD(SYSLST)	Yes, for filename	Messages are routed to the file defined for filename.
	Yes, for SYSLST	COBOL/VSE DISPLAY data is routed to SYSLST.
	No	If the file designated by filename is not defined when output is directed to it, the LE/VSE condition with symbolic feedback code CEE0E9 is raised. If SYSLST is not defined when output is directed to it, the LE/VSE condition with symbolic feedback code CEE0E9 is raised.
MSGFILE(filename_1) OUTDD(filename_2)	Yes, for filename_1	Messages are routed to the file defined for filename_1.
	Yes, for filename_2.	COBOL/VSE DISPLAY data is routed to filename_2.
	No	If the file designated by filename_1 is not defined when output is directed to it, the LE/VSE condition with symbolic feedback code CEE0E9 is raised. If the file designated by filename_2 is not defined when output is directed to it, the LE/VSE condition with symbolic feedback code CEE0E9 is raised.

Note:

1. DOS/VS COBOL run-time messages, DOS/VS COBOL DISPLAY data, and DISPLAY data produced by VS COBOL II routines compiled with the VS COBOL II compiler under VSE are always directed to SYSLST.

For more information about directing COBOL output, refer to *IBM COBOL for VSE/ESA Programming Guide*

Using PL/I Input/Output Statements

LE/VSE directs run-time messages in PL/I routines to the file specified by the LE/VSE MSGFILE run-time option, instead of to the PL/I SYSPRINT STREAM PRINT file.

User-specified output is still directed to the PL/I SYSPRINT STREAM PRINT file. If you want LE/VSE to handle this output, specify the run-time option MSGFILE(SYSPRINT). When you use MSGFILE(SYSPRINT), LE/VSE routes all output directed to the PL/I SYSPRINT STREAM PRINT file to SYSLST, unless the SYSPRINT file constant declaration includes the INTERNAL attribute. If the INTERNAL attribute is specified in the SYSPRINT declaration, user-specified output is routed to the device specified in the MEDIUM option of the ENVIRONMENT attribute.

When you specify MSGFILE(SYSPRINT):

- Run-time messages and user-specified output directed to the PL/I SYSPRINT STREAM PRINT EXTERNAL file are routed to SYSLST, using the attributes shown in Table 37 on page 199.
- Any file constant declaration that includes SYSPRINT STREAM PRINT EXTERNAL file attributes is ignored.
- Any OPENs and CLOSEs to the PL/I SYSPRINT STREAM PRINT EXTERNAL file are ignored.
- If SYSLST is not defined when output is first directed to it, the LE/VSE condition with the symbolic feedback code CEE0E9 is raised.
- Synchronization between the types of output (messages and user-specified output) is not provided, so the order of the output is unpredictable.

MSGFILE Considerations When Using PL/I

If MSGFILE(SYSPRINT) is in effect, use SYSPRINT only to direct output to the PL/I SYSPRINT STREAM PRINT file.

Because performance might be degraded with the MSGFILE(SYSPRINT) option, it is recommended only for debugging purposes. For production applications, do not use the MSGFILE(SYSPRINT) option if your applications direct user-created output to the PL/I SYSPRINT STREAM PRINT file.

In the batch environment, nested enclaves in an LE/VSE process can share the same PL/I SYSPRINT STREAM PRINT EXTERNAL file. In a nested enclave environment, if you want to use MSGFILE(SYSPRINT) you should specify MSGFILE(SYSPRINT) for all enclaves in the application that contain SYSPRINT PUT statements. When you specify MSGFILE(SYSPRINT), the file is opened by LE/VSE when run-time message or user-specified output is first directed to it, and is closed by LE/VSE at process termination.

Under CICS, the MSGFILE run-time option defaults to the CESE transient data queue. Both run-time messages and the SYSPRINT STREAM PRINT EXTERNAL file output are directed to this transient data queue. The CESE transient data queue is a CICS thread-level resource. For more information about message handling and run-time message output under CICS, see “Run-Time Output under CICS” on page 308.

For more information about directing PL/I output, refer to *IBM PL/I for VSE/ESA Programming Guide*.

Examples Using Multiple Message Handling Callable Services

The examples in this section show how to use the LE/VSE message and condition-handling services to issue a message that relates to a condition token. The same calls are illustrated in C, PL/I, and COBOL.

Each example illustrates how CEEMOUT dispatches an informational message and uses CEENCOD to construct a token for the message. The message area is then initialized, CEEMGET retrieves the message, and CEEDCOD decodes the feedback token from CEEMGET. After all of the message has been retrieved, CEEMOUT issues the message. If any of the services fail, CEEMSG issues an informational error message.

C Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG

```
/*Module/File Name:  EDCMSGSX */
 /************************************************************
 **FUNCTION  : CEEMOUT - dispatch a message to message file  *
 **          : CEENCOD - construct a condition token        *
 **          : CEEMGET - retrieve, format and store a message*
 **          : CEEDCOD - decode an existing condition token  *
 **          : CEEMSG  - retrieve, format, and dispatch a    *
 **          :         - message to message file            *
 **                                                          *
 **   This example illustrates the invocation of the LE/VSE  *
 **   Message and Condition Handling services.               *
 **   It constructs a condition token, retrieves the         *
 **   associated message, and outputs the message to the     *
 **   message file.                                          *
 **                                                          *
 **   This example program will output the LE/VSE            *
 **   message,"CEE0260S".                                    *
 *************************************************************/
#include <string.h>
#include <stdio.h>
#include <leawi.h>
#include <stdlib.h>
#include <ceeedcct.h>

int main(void) {

  _VSTRING message;
  _INT4 dest,msgindx;
  _CHAR80 msgarea;
  _FEEDBACK fc,token;
  _INT2 c_1,c_2,cond_case,sev,control;
  _CHAR3 facid;
  _INT4 isi;

  printf ( "\n**********************************\n");
  printf ( "\nCE92MSG C Example is now in motion\n");
  printf ( "\n**********************************\n");

  strcpy(message.string,"The following message, CEE0260S, is expected");
  message.length = strlen(message.string);
  dest = 2;
```

Figure 71. C Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG (Part 1 of 2)

```
  /**************************************************************
   *   Call CEEMOUT to output informational message.          *
   *   Call CEEMSG to output error message if CEEMOUT fails.   *
   ************************************************************/
  CEEMOUT(&message,&dest,&fc);

  if ( _FBCHECK (fc , CEE000) != 0 ) {
     /* put the message if CEEMOUT failed */
     dest = 2;
     CEEMSG(&fc,&dest,NULL);
     exit(2999);
  }
  /*****************************************
   * Construct a to ken for CEE message 0260.*
   *****************************************/
  c_1 = 3;
  c_2 = 260;
  cond_case = 1;
  sev = 3;
  control = 1;
  memcpy(facid,"CEE",3);
  isi = 0;

  CEENCOD(&c_1,&c_2,&cond_case,&sev,&control,
          facid,&isi,&token,&fc);
  if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
     printf("CEENCOD failed with message number %d\n",
            fc.tok_msgno);
     exit(2999);
  }

  /*****************************************
   * Initialize the message area.          *
   *****************************************/
  msgindx = 0;
  memset(msgarea,' ',79);
  msgarea[80] = '\0';

  /*****************************************************************
   * Use CEEMGET until all the message has been retrieved.        *
   * Msgindx will be zero when all the message has been retrieved.*
   * Call CEEMSG to output error message if CEEMGET fails.        *
   ***************************************************************/
  do {
     CEEMGET(&token,msgarea,&msgindx,&fc);
     if (fc.tok_sev > 1) {
        dest = 2;
        CEEMSG(&fc,&dest,NULL);
        exit(2999);
     }
     memcpy(message.string,msgarea,80);
     message.length = 80;
     dest = 2;

     CEEMOUT(&message,&dest,&fc);          /* put out the message */

     if ( _FBCHECK (fc , CEE000) != 0 ) {
        dest = 2;
        CEEMSG(&fc,&dest,NULL);
        exit(2999);
     }
  } while (msgindx != 0);
  printf ( "\n**********************************\n");
  printf ( "\nCE92MSG C Example is now ended    \n");
  printf ( "\n**********************************\n");
}
```

Figure 71. C Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG (Part 2 of 2)

COBOL Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and CEEMSG

```
CBL LIB,APOST
      *Module/File Name: IGZTMSGS
      *************************************************************
      *                                                           *
      *  CE92MSG - Program to invoke the following LE services:   *
      *                                                           *
      *          : CEEMOUT - dispatch a message to message file   *
      *          : CEENCOD - construct a condition token          *
      *          : CEEMGET - retrieve, format and store a message *
      *          : CEEDCOD - decode an existing condition token   *
      *          : CEEMSG  - retrieve, format, and dispatch a     *
      *          :           message to message file              *
      *                                                           *
      * This example illustrates the invocation of the Language   *
      * Environment Message and Condition Handling services.      *
      * It constructs a condition token, retrieves the associated *
      * message, and outputs the message to the message file.     *
      *                                                           *
      * This example program will output the Language Environment *
      * message, 'CEE0260S'.                                      *
      *                                                           *
      *************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CE92MSG.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  MSGSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of MSGSTR.
       01  MSGDEST                 PIC S9(9) BINARY.
       01  SEV                     PIC S9(4) BINARY.
       01  MSGNO                   PIC S9(4) BINARY.
       01  CASE                    PIC S9(4) BINARY.
       01  SEV2                    PIC S9(4) BINARY.
       01  CNTRL                   PIC S9(4) BINARY.
       01  FACID                   PIC X(3).
       01  ISINFO                  PIC S9(9) BINARY.
       01  MSGINDX                 PIC S9(9) BINARY.
       01  CTOK.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) BINARY.
```

Figure 72. COBOL Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and CEEMSG (Part 1 of 3)

```
01  FC.
       02  Condition-Token-Value.
       COPY  CEEIGZCT.
           03  Case-1-Condition-ID.
               04  Severity    PIC S9(4) BINARY.
               04  Msg-No      PIC S9(4) BINARY.
           03  Case-2-Condition-ID
                     REDEFINES Case-1-Condition-ID.
               04  Class-Code  PIC S9(4) BINARY.
               04  Cause-Code  PIC S9(4) BINARY.
           03  Case-Sev-Ctl    PIC X.
           03  Facility-ID     PIC XXX.
       02  I-S-Info            PIC S9(9) BINARY.
   01  MGETFC.
       02  Condition-Token-Value.
       COPY  CEEIGZCT.
           03  Case-1-Condition-ID.
               04  Severity    PIC S9(4) BINARY.
               04  Msg-No      PIC S9(4) BINARY.
           03  Case-2-Condition-ID
                     REDEFINES Case-1-Condition-ID.
               04  Class-Code  PIC S9(4) BINARY.
               04  Cause-Code  PIC S9(4) BINARY.
           03  Case-Sev-Ctl    PIC X.
           03  Facility-ID     PIC XXX.
       02  I-S-Info            PIC S9(9) BINARY.
   01  MSGAREA                 PIC X(80).
   PROCEDURE DIVISION.
   0001-BEGIN-PROCESSING.
       DISPLAY '**************************************'.
       DISPLAY 'CE92MSG COBOL Example is now in motion. '.
       DISPLAY '**************************************'.
       MOVE 80 TO Vstring-length of MSGSTR.
       MOVE 'The following error message, CEE0260S, is expected:'
               TO Vstring-text of MSGSTR.
       MOVE 2 TO MSGDEST.
  **************************************************************
  **   Call CEEMOUT to put out informational message.        **
  **************************************************************
       CALL 'CEEMOUT' USING MSGSTR , MSGDEST , FC.
       IF NOT CEE000 of FC  THEN
           DISPLAY 'Error ' Msg-No of FC
              ' in issuing header message'
           STOP RUN
       END-IF.
  **************************************************************
  **   Set up token fields for creation of a condition token  **
  **************************************************************
       MOVE   3 TO SEV.
       MOVE 260 TO MSGNO.
       MOVE   1 TO CASE.
       MOVE   3 TO SEV2.
       MOVE   1 TO CNTRL.
       MOVE 'CEE' TO FACID.
       MOVE   0 TO ISINFO.
  **************************************************************
  **   Call CEENCOD to construct a condition token            **
  **************************************************************
       CALL 'CEENCOD' USING SEV, MSGNO, CASE, SEV2, CNTRL,
           FACID, ISINFO, CTOK, FC.
       IF CEE000 of FC THEN
           MOVE 0 TO MSGINDX
           MOVE SPACES TO MSGAREA
```

Figure 72. COBOL Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and CEEMSG (Part 2 of 3)

```
*************************************************************
  **      Call CEEMGET to retrieve message 260. Since       **
  **      message 260 is longer than the length of MSGAREA, **
  **      a PERFORM statement loop is used to call CEEMGET   **
  **      multiple times until the message index is zero.    **
  *************************************************************
          PERFORM TEST AFTER UNTIL( MSGINDX = 0 )
              CALL 'CEEMGET' USING CTOK, MSGAREA, MSGINDX, MGETFC
              IF (MGETFC NOT = LOW-VALUE) THEN
  *************************************************************
  *          Call CEEDCOD to decode CEEMGET's feedback token **
  *************************************************************
                  CALL 'CEEDCOD' USING MGETFC, SEV, MSGNO,
                      CASE, SEV2, CNTRL, FACID, ISINFO, FC
                  IF NOT CEE000 of FC  THEN
  *************************************************************
  *          Call CEEMSG to output LE error message         **
  *          using feedback code from CEEDCOD call.         **
  *************************************************************
                      CALL 'CEEMSG' USING MGETFC, MSGDEST, FC
                      IF NOT CEE000 of FC THEN
                          DISPLAY 'Error ' Msg-No of FC
                          ' from CEEMSG after error in CEEDCOD'
                      END-IF
                      STOP RUN
                  END-IF
  *************************************************************
  *               If decoded message number is not 455,     **
  *               then CEEMGET actually failed with error.  **
  *************************************************************
                  IF ( Msg-No of MGETFC NOT = 455) THEN
                      DISPLAY 'Error ' Msg-No of MGETFC
                          ' retrieving message CEE0260S'
                      STOP RUN
                  END-IF
              END-IF
  *************************************************************
  *      Call CEEMOUT to output earch portion of message 260 **
  *************************************************************
              MOVE MSGAREA TO Vstring-text of MSGSTR
              CALL 'CEEMOUT' USING MSGSTR , MSGDEST , FC
              IF (MSGINDX = ZERO) THEN
                  DISPLAY '*************************************'
                  DISPLAY ' Cobol message example program ended.'
                  DISPLAY '*************************************'
              END-IF
          END-PERFORM
      ELSE
          DISPLAY 'Error ' Msg-No of FC
             ' in encoding condition token'
          STOP RUN
      END-IF.

      GOBACK.
```

Figure 72. COBOL Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and CEEMSG (Part 3 of 3)

PL/I Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG

```
*PROCESS MACRO;
    /*Module/File Name: IBMMSGS                                */
    /*************************************************************/
    /*                                                           */
    /* FUNCTION : CEEMOUT - dispatch a message to message file   */
    /*          : CEENCOD - construct a condition token          */
    /*          : CEEMGET - retrieve, format and store a message */
    /*          : CEEDCOD - decode an existing condition token   */
    /*          : CEEMSG  - retrieve, format, and dispatch a     */
    /*                      message to message file              */
    /*                                                           */
    /*   This example illustrates the invocation of the LE/VSE   */
    /*   Message and Condition Handling services.                */
    /*   It constructs a condition token, retrieves the          */
    /*   associated message, and outputs the message to the      */
    /*   message file.                                           */
    /*                                                           */
    /*   This example program will output the LE/VSE             */
    /*   message, "CEE0260S"                                     */
    /*                                                           */
    /*************************************************************/
 CE92MSG: PROC OPTIONS(MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL MSGSTR    CHAR(255) VARYING;
    DCL MSGDEST   INT4;
    DCL MSGNUM    INT2;
    DCL CASE      INT2;
    DCL SEV       INT2;
    DCL SEV2      INT2;
    DCL CNTRL     INT2;
    DCL FACID     CHARACTER ( 3 );
    DCL ISINFO    INT4;
    DCL MSGINDX   INT4;
    DCL 01 CTOK   FEEDBACK;
    DCL 01 FC     FEEDBACK;
    DCL 01 MGETFC FEEDBACK;
    DCL MSGAREA   CHAR(80);

    PUT SKIP LIST('PL/I message example is now in motion');
    MSGSTR = 'The following message, CEE0260S, is expected';
    MSGDEST = 2;
    /*************************************************************/
    /*   Call CEEMOUT to output informational message.           */
    /*   Call CEEMSG to output error message if CEEMOUT fails.   */
    /*************************************************************/
    CALL CEEMOUT ( MSGSTR, MSGDEST, FC );
    IF  &#170; FBCHECK( FC, CEE000 )  THEN
       CALL CEEMSG( FC, MSGDEST, MGETFC );
    /*************************************************************/
    /*   Set up token fields for creation of a condition token   */
    /*************************************************************/
    SEV = 3;
    MSGNUM = 260;
    CASE = 1;
    SEV2 = 3;
    CNTRL = 1;
    FACID = 'CEE';
    ISINFO = 0;
```

Figure 73. PL/I Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG (Part 1 of 2)

```
  /*************************************************************/
 /*   Call CEENCOD to construct a condition token            */
 /*************************************************************/
 CALL CEENCOD ( SEV, MSGNUM, CASE, SEV2, CNTRL, FACID,
    ISINFO, CTOK, FC );
 IF  FBCHECK( FC, CEE000)   THEN  DO;
    MSGINDX = 0;
    MSGAREA = ' ';
    /*********************************************************/
    /* Call CEEMGET to retrieve message 260. Since           */
    /* message 260 is longer than the length of MSGAREA,     */
    /* a DO UNTIL statement loop is used to call CEEMGET     */
    /* multiple times until the message index is zero.       */
    /*********************************************************/
Retrieve_Message:
    DO UNTIL( MSGINDX = 0 );
       CALL CEEMGET ( CTOK, MSGAREA, MSGINDX, MGETFC );
       IF  &#170; FBCHECK( MGETFC, CEE000)  THEN  DO;
          /**************************************************/
          /*Call CEEDCOD to decode CEEMGET's feedback token   */
          /**************************************************/
          CALL CEEDCOD ( MGETFC, SEV, MSGNUM,
             CASE, SEV2, CNTRL, FACID, ISINFO, FC );
          IF  &#170; FBCHECK( FC, CEE000)  THEN  DO;
             /***********************************************/
             /* Call CEEMSG to output LE/VSE error message    */
             /* associated with feedback token from CEEMGET.  */
             /***********************************************/
             CALL CEEMSG  ( MGETFC, MSGDEST, FC );
             IF  &#170; FBCHECK( FC, CEE000)  THEN  DO;
                PUT SKIP LIST ('Error ' || FC.MsgNo
                   || ' from CEEMSG');
                STOP;
                END;
             /***********************************************/
             /* If decoded message number is not 455,         */
             /* then CEEMGET actually failed with error.      */
             /***********************************************/
             IF ( MGETFC.MsgNo &#170;= 455)  THEN  DO;
                PUT SKIP LIST( 'Error ' || MGETFC.MsgNo
                      || ' retrieving message CEE0260S');
                STOP;
                END;
             END;
          END;
       /*****************************************************/
       /* Call CEEMOUT to output each portion of message 260  */
       /*****************************************************/
       MSGSTR = MSGAREA;
       CALL CEEMOUT ( MSGSTR, MSGDEST, FC );
       IF (MSGINDX = 0) THEN  DO;
          PUT SKIP LIST ('**********************************');
          PUT SKIP LIST ('PL/I message example program ended');
          PUT SKIP LIST ('**********************************');
          END;
       END Retrieve_Message /* END DO UNTIL MSGINDX = 0 */;
    END /* CEENCOD successful */;
 ELSE  DO;
    PUT SKIP LIST ('Error ' || FC.MsgNo
       || ' in encoding condition token');
    END;

END CE92MSG;
```

Figure 73. PL/I Example Illustrating Calls to CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, and CEEMSG (Part 2 of 2)

Chapter 16. Using Date and Time Services

This chapter describes LE/VSE date and time services and includes examples showing calls to those services.

Understanding the Basics

LE/VSE includes a complete set of callable services that help HLLs perform date and time calculations. You can use these services to read, calculate, and write values representing the date and time. LE/VSE offers unique pattern-matching capabilities that let you process almost any date and time format contained in an input record or produced by operating system services.

You can use date and time services to:
- Format date and time values by country code
- Format date and time values using customized formats
- Parse date values and time values
- Convert between Gregorian, Julian, Asian, and Lilian formats
- Calculate days between dates
- Calculate elapsed time to the nearest millisecond
- Get local time and *Greenwich Mean Time* (GMT) from the system without a *supervisor call* (SVC) overhead
- Properly handle 2-digit years in the year 2000 and beyond

All LE/VSE date and time services are enabled for national language support, including full DBCS support for the Japanese Emperor and Chinese eras. For more information on national language support see Chapter 17, "National Language Support Services," on page 247.

All LE/VSE date and time services are based on the Gregorian calendar, with Lilian limits as described in "Date Limits" on page 214.

© Copyright IBM Corp. 1991, 2005

Related Services

Callable Services

CEECBLDY Converts character date value to the COBOL Integer format. Day one is 01 January 1601 and the value is incremented by one for each subsequent day. This service is similar to CEEDAYS, except that it provides an answer in COBOL Integer format, so that it is compatible with ANSI COBOL intrinsic functions. It should not be used with other LE/VSE date or time services.

CEEDATE Converts dates in the Lilian format to character values

CEEDATM Converts number of seconds to character timestamp

CEEDAYS Converts character date values to the Lilian format. Day one is 15 October 1582, and the value is incremented by one for each subsequent day.

CEEDYWK Provides day of week calculation

CEEGMT Gets current Greenwich Mean Time (date and time)

CEEGMTO Gets difference between Greenwich Mean Time and local time

CEEISEC Converts binary year, month, day, hour, minute, second, and millisecond to a number representing the number of seconds since 00:00:00 15 October 1582

CEELOCT Gets current date and time

CEEQCEN Queries the century window

CEESCEN Sets the century window

CEESECI Converts a number representing the number of seconds since 00:00:00 15 October 1582 to seven separate binary integers representing year, month, day, hour, minute, second, and millisecond

CEESECS Converts character timestamps (a date and time) to the number of seconds since 00:00:00 15 October 1582

CEEUTC Same as CEEGMT

See *LE/VSE Programming Reference* for further information about these callable services.

Working with Date and Time Services

Before you can start working with date and time services, you need to know the various formats for specifying date and times and any limits that exist.

Date Limits

All LE/VSE date and time services are based on the Gregorian calendar, which has certain limits for the date variables. These limits are:

Starting Lilian Date (Non-COBOL)
The beginning of the valid Lilian date range (day one) is Friday, 15

October 1582, the date the Gregorian calendar was adopted. Lilian dates preceding this date are undefined. In the Lilian date range:
Day zero equals 00:00:00 14 October 1582.
Day one equals 00:00:00 15 October 1582.

All valid Lilian dates must be after 00:00:00 15 October 1582.

Starting COBOL Integer Date (COBOL Intrinsic Functions)
The beginning of the COBOL Integer date range according to the COBOL standard is 31 December 1600. COBOL Integer dates preceding this date are undefined. In the COBOL Integer date range:
Day zero equals 00:00:00 31 December 1600.
Day one equals 00:00:00 01 January 1601.

All valid COBOL Integer dates must be after 00:00:00 01 January 1601.

End Lilian Date (End COBOL Integer Date)
The end of the Lilian date range, as well as the COBOL Integer date range, is set to 31 December 9999. Lilian dates and COBOL Integer dates following this date are undefined.

Limit of Current Era
The maximum future date you can express in an era system (Japanese Era or Chinese Era) must be within the first 999 years of the current era. Future dates past year 999 of the current era are undefined.

Picture Character Terms and Picture Strings

Picture character terms define the format of date and time fields. A picture string is a template that indicates the format of the input data. For example, the format of the date 06/16/1990 (where 06 is the month, 16 is the day, and 1990 is the year) corresponds to the picture string MM/DD/YYYY. See *LE/VSE Programming Reference* for the LE/VSE picture character term and picture string values.

Notation for Eras

Calendars based on eras use unique picture strings to identify the eras. The era picture string begins with a less than character (<) and ends with the greater than character (>). The characters between the less than and greater than characters are the Japanese era name in DBCS characters or Chinese era name in DBCS characters. The picture strings identify the eras as follows:

Japanese Era The six-character string <JJJJ>. An example of specifying the Japanese Meiji era would be to specify X'0E45A645840F' where the X'0E' and X'0F' are the less than character (<) and greater than character (>), respectively. Refer to *LE/VSE Programming Reference* for the Japanese Eras used by LE/VSE date and time services.

Chinese Era The six-character string <CCCC> or the ten-character string <CCCCCCCC>. An example of specifying the MinKow era would be to specify X'0E4DB256CE0F' where the X'0E' and X'0F' are the less than character (<) and greater than character (>), respectively. Refer to *LE/VSE Programming Reference* for the Chinese Eras used by LE/VSE date and time services.

Performing Calculations on Date and Time Values

LE/VSE stores a date as a fullword binary integer and a timestamp as a doubleword floating-point value. You can use these formats to perform arithmetic calculations on date and time values, instead of writing special subroutines to do so. Figure 74 is an example of how you can use LE/VSE date and time services to convert a date to a different format and perform a simple calculation on the formatted date.

In this example, the number of years of service for an employee is determined using the original date of hire in the format YYMMDD to make the calculations. The example calculates the total number of years of service for an employee by first calling CEEDAYS to convert the days to Lilian and by then calling CEELOCT (Get Current Local Time) to get the current local time. Then, *doh_Lilian* is subtracted from *today_Lilian* (the number of days from the beginning of the Gregorian calendar to the current local time) to calculate the employee's total number of days of employment. The final calculation divides that number by 365.25 to get the number of service years.

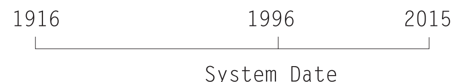
```
CALL CEEDAYS (date_of_hire, 'YYMMDD', doh_Lilian, fc)
CALL CEELOCT (today_Lilian, today_seconds, today_Gregorian, fc)
service_days = today_Lilian - doh_Lilian
service_years = service_days / 365.25
```

Figure 74. Performing Calculations on Dates

The valid Lilian date range is 15 October 1582 to 31 December 9999. However, COBOL intrinsic functions uses the COBOL Integer date 01 January 1601 as day one. LE/VSE provides the CEECBLDY callable service to allow you to work with the COBOL Integer date format. See *LE/VSE Programming Reference* for more information on the CEECBLDY service.

Century Window Routines

To process 2-digit years in the year 2000 and beyond, LE/VSE employs a sliding scheme called a *century window* where all 2-digit years lie within a 100-year interval. The default century window for LE/VSE is set to start 80 years before the current system date. In the following example, 1996 is the current system date. The century window spans one hundred years from 1916 to 2015 where years 16 through 99 are recognized as 1916-1999 and years 00 through 15 are recognized as 2000-2015.



In 1997, years 17 through 99 are recognized as 1917-1999, and years 00 through 16 are recognized as 2000-2016.

By year 2080, all 2-digit years would be recognized as 20*xx*. In 2081, 00 would be recognized as year 2100.

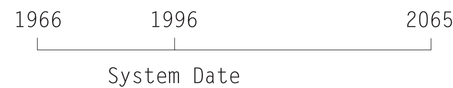
Note: If you use the DATE job control statement to override the system date, LE/VSE calculates the century window based upon the year specified in the DATE job control statement.

Some applications might need to set up a different 100-year interval. For example, banks often deal with 30-year bonds, which could be due 01/31/20. You can use the CEESCEN callable service (see *LE/VSE Programming Reference*) to change the century window.

For example,

```
Call CEESCEN(30, fc)
```

sets the default century to the 100-year interval starting 30 years prior to the system date, instead of the LE/VSE default of 80 years.



A companion service, CEEQCEN, queries the current century window. A subroutine can, for example, use a different interval for date processing than the parent routine. Before returning, the subroutine resets the interval back to its previous value.

For more information about changing the century window, see “Examples Illustrating Calls to CEEQCEN and CEESCEN” on page 219.

National Language Support for Date and Time Services

The NATLANG and COUNTRY run-time options provide national language support for date and time services. The names of the months and days of the week are based on the national language specified in the NATLANG option. In addition, some date and time services allow the specification of a blank or null picture string, a practice that directs LE/VSE to use a date and time format based upon the current value specified in the COUNTRY option. You can locate the default date and time format for any supported country by using the CEEFMDA, CEEFMDT, or CEEFMTM callable services.

Examples Using Date and Time Callable Services

The examples in this section illustrate some of the date conversion and manipulation you can perform by using the LE/VSE date and time services together. There are examples for the following services:

- CEEQCEN—Query the century window (see “Examples Illustrating Calls to CEEQCEN and CEESCEN” on page 219)
- CEESCEN—Set the century window (see “Examples Illustrating Calls to CEEQCEN and CEESCEN” on page 219)
- CEESECS—Convert timestamp to seconds (see “Examples Illustrating Calls to CEESECS” on page 222)
- CEESECS and CEEDATM—Convert timestamp to seconds and build a new timestamp (see “Examples Illustrating Calls to CEESECS and CEEDATM” on page 226)
- CEESECS, CEESECI, CEEISEC, and CEEDATM—Convert timestamp to seconds, convert seconds to date and time components, convert date and time to seconds, and build new timestamp (see “Examples Illustrating Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM” on page 231)

- CEEDAYS, CEEDYWK, and CEEDATE—Convert a date to a Lilian date, convert Lilian date to calendar format, and return day of week for the derived Lilian date (see “Example Illustrating Calls to CEEDAYS, CEEDATE, and CEEDYWK” on page 238)
- CEECBLDY—Converts a date to a COBOL Integer date so that it is compatible with ANSI COBOL intrinsic functions (see “Calls to CEECBLDY in COBOL” on page 245)

Examples Illustrating Calls to CEEQCEN and CEESCEN

The following examples illustrate how to query the current century window and how to set a new window with a new default of 30 years.

Calls to CEEQCEN and CEESCEN in C

```
/*Module/File Name:  EDCCWIN  */
/*****************************************************************/
/* Demonstrates how to use CEEQCEN and CEESCEN to query and      */
/* set the century window.                                       */
/*****************************************************************/

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <leawi.h>
#include <ceeedcct.h>

int main (void) {

_INT4 oldcen, tempcen;
_FEEDBACK qcenfc,scenfc;

/* Call CEEQCEN to retrieve and save current century window  */
  CEEQCEN ( &oldcen , &qcenfc );
  if ( _FBCHECK ( qcenfc , CEE000 ) != 0 ) {
     printf("CEEQCEN failed with message number %d\n",
            qcenfc.tok_msgno);
     exit(1999);
  }

 /* Call CEESCEN to temporarily change century window to 30  */
  tempcen = 30;
  CEESCEN ( &tempcen , &scenfc );
  if ( _FBCHECK ( scenfc , CEE000 ) != 0 ) {
    printf(
    "CEESCEN (1st call) failed with message number %d\n",
           scenfc.tok_msgno);
    exit(2999);
  }

/* Perform date processing with 2-digit years...            */

/* .
   .
   . */

 /* Call CEESCEN again to reset century window              */
  CEESCEN ( &oldcen , &scenfc );
  if ( _FBCHECK ( scenfc , CEE000 ) != 0 ) {
    printf(
    "CEESCEN (2nd call) failed with message number %d\n",
           scenfc.tok_msgno);
    exit(3999);
  }
  exit (0);
}
```

Figure 75. C Example of Querying and Changing the Century Window

Calls to CEEQCEN and CEESCEN in COBOL

```
CBL LIB,APOST
      *Module/File Name: IGZTCWIN
      ****************************************************************
      *                                                              *
      * Demonstrates how to use CEEQCEN and CEESCEN to query         *
      * and set the century window.                                  *
      *                                                              *
      ****************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  CBCENTW.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77  OLDCEN   PIC S9(9) BINARY.
       77  TEMPCEN  PIC S9(9) BINARY.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity       PIC S9(4) BINARY.
                   04  Msg-No         PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code     PIC S9(4) BINARY.
                   04  Cause-Code     PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info       PIC S9(9) BINARY.

       PROCEDURE DIVISION.
      **************************************************************
      ** Call CEEQCEN to retrieve and save current century window **
      **************************************************************
           CALL 'CEEQCEN' USING OLDCEN , FC.
           IF NOT CEE000 of FC  THEN
              DISPLAY 'CEEQCEN failed with msg '
                   Msg-No of FC UPON CONSOLE
              STOP RUN
           END-IF.

      **************************************************************
      ** Call CEESCEN to temporarily change century window to 30  **
      **************************************************************
           MOVE 30 TO TEMPCEN.
           CALL 'CEESCEN' USING TEMPCEN , FC.
           IF NOT CEE000 of FC  THEN
              DISPLAY 'First call to CEESCEN failed with msg '
                  Msg-No of FC UPON CONSOLE
              STOP RUN
           END-IF.

      ** Perform date processing with 2-digit years...
      *       .
      *       .
      *       .
      ** Call CEESCEN again to reset century window

           CALL 'CEESCEN' USING OLDCEN , FC.
           IF NOT CEE000 of FC  THEN
              DISPLAY 'Second call to CEESCEN failed with msg '
                 Msg-No of FC UPON CONSOLE
              STOP RUN
           END-IF.

           GOBACK.
```

Figure 76. COBOL Example of Querying and Changing the Century Window

Calls to CEEQCEN and CEESCEN in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMCWIN                           */
 /****************************************************/
 /*                                                   */
 /* Demonstrates how to use CEEQCEN and               */
 /* CEESCEN to query and set the century window.      */
 /* window.                                           */
 /*                                                   */
 /****************************************************/
 PLCENTW: PROC OPTIONS (MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL OLDCEN  INT4;
    DCL TEMPCEN INT4;
    DCL 01 FC   FEEDBACK;

    /* Call CEEQCEN to retrieve and save current century window  */
    CALL CEEQCEN (OLDCEN, FC);
    IF &#170; FBCHECK( FC, CEE000)  THEN  DO;
       DISPLAY( 'CEEQCEN failed with msg '|| FC.MsgNo );
       STOP;
       END;

    /* Call CEESCEN to temporarily change century window to 30  */
    TEMPCEN = 30;
    CALL CEESCEN (TEMPCEN, FC);
    IF &#170; FBCHECK( FC, CEE000)  THEN  DO;
       DISPLAY( 'First call to CEESCEN failed with msg '
          || FC.MsgNo );
       STOP;
       END;
    /* Perform date processing with 2-digit years...           */
    /* .
       .
       .                                                       */

    /* Call CEESCEN again to reset century window              */
    CALL CEESCEN (OLDCEN, FC);
    IF &#170; FBCHECK( FC, CEE000)  THEN  DO;
       DISPLAY( 'Second call to CEESCEN failed with msg '
          || FC.MsgNo );
       STOP;
       END;

 END PLCENTW;
```

Figure 77. PL/I Example of Querying and Changing the Century Window

Examples Illustrating Calls to CEESECS

The following examples illustrate calls to CEESECS to compute the total number of hours between two timestamps.

Calls to CEESECS in C

```
/*Module/File Name:  EDCDT1   */
 /*********************************************************************/
 /*Function       : CEESECS - convert timestamp to seconds            */
 /*This example calls the LE CEESECS callable service to compute the hour*/
 /* number of numbers between the timestamps 11/02/92 05:22 and         */
 /* 11/02/92 17:22. The program responds that 36 hours has elapsed.      */
 /*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>
main ()
{
  _VSTRING StartTime;
  _VSTRING EndTime;
  _VSTRING  picstr;
  _FLOAT8 Start_Secs;
  _FLOAT8 End_Secs;
  _FLOAT8 Elapsed_Time;
  _FEEDBACK FC;
  _INT4 dest=2;
  /***************************************************************
   The date picstr must be set to match the timestamp format.
   **************************************************************/
   strncpy (picstr.string,"MM/DD/YY HH:MI",14);
   picstr.length = 14;
   strncpy(StartTime.string,"11/02/92 05:22",14);
   StartTime.length = 14;
   strncpy(EndTime.string,"11/03/92 17:22",14);
   EndTime.length = 14;
  /**************************************************************
   CEESECS takes the start time and returns
   a double-precision Lilian seconds tally in Start_Secs.
   **************************************************************/
   CEESECS ( &StartTime, &picstr , &Start_Secs , &FC );
   if ( _FBCHECK (FC , CEE000) == 0 )
   {
  /**************************************************************
   CEESECS takes the end time and returns
   a double-precision Lilian seconds tally in End_Secs.
   **************************************************************/
     CEESECS ( &EndTime, &picstr , &End_Secs , &FC );
     if ( _FBCHECK (FC , CEE000) == 0 )
     {
       Elapsed_Time = (End_Secs - Start_Secs)/3600.0;
       printf("%4.2f hours have elapsed between %s and %s.\n",
           Elapsed_Time, StartTime.string, EndTime.string);
     }
     else
     {
       printf ( "Error converting TimeStamp to seconds.\n" );
       CEEMSG(&FC, &dest, NULL);
     }
   }
   else
   {
    printf ( "Error converting TimeStamp to seconds.\n" );
    CEEMSG(&FC, &dest, NULL);
   }
}
```

Figure 78. Calls to CEESECS in C

Calls to CEESECS in COBOL

```
CBL LIB,APOST
      *Module/File Name: IGZTDT1
      *********************************************************
      **                                                     **
      ** CEE78DAT - Call CEESECS to convert timestamp to    **
      **            seconds                                  **
      **                                                     **
      ** This example calls the LE CEESECS callable         **
      **  service to compute the number of hours between    **
      **  the timestamps 11/02/92 05:22 and 11/02/92 17:22. **
      **  The program responds that 36 hours has elapsed.   **
      **                                                     **
      *********************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CE78DAT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      *  Double precision is needed for the seconds results
       01  START-SECS             COMP-2.
       01  END-SECS               COMP-2.
       01  EOF-SWITCH             PIC X VALUE 'N'.
           88  EOF                VALUE 'Y'.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity       PIC S9(4) BINARY.
                   04  Msg-No         PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code     PIC S9(4) BINARY.
                   04  Cause-Code     PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info       PIC S9(9) BINARY.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char       PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of PICSTR.
       01  START-TIME.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char       PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of START-TIME.
       01  END-TIME.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char       PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of END-TIME.
       01  INPUT-VARIABLES.
           05  ELAPSED-TIME        PIC S9(5)V99 PACKED-DECIMAL.
           05  ELAPSED-TIME-OUT    PIC +Z(4)9.99.

       PROCEDURE DIVISION.
```

Figure 79. Calls to CEESECS in COBOL (Part 1 of 2)

```
 0001-BEGIN-PROCESSING.
     MOVE  14 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY HH:MI' TO Vstring-text of PICSTR.
     MOVE  14 TO Vstring-length of START-TIME.
     MOVE '11/02/92 05:22' TO Vstring-text of START-TIME.
     MOVE  14 TO Vstring-length of END-TIME.
     MOVE '11/03/92 17:22' TO Vstring-text of END-TIME.
*    *********************************************************
*    * CEESECS takes the timestamp START-TIME and returns a  *
*    * double-precision Lilian seconds tally in START-SECS.  *
*    *********************************************************
     CALL 'CEESECS' USING START-TIME, PICSTR, START-SECS, FC
     IF CEE000 of FC THEN
*        *****************************************************
*        * CEESECS takes the timestamp END-TIME and returns a  *
*        * double-precision Lilian seconds tally in END-SECS.  *
*        *****************************************************
         CALL 'CEESECS' USING END-TIME, PICSTR, END-SECS, FC
         IF CEE000 of FC THEN
             COMPUTE ELAPSED-TIME = (END-SECS - START-SECS) / 3600
             MOVE ELAPSED-TIME TO ELAPSED-TIME-OUT
             DISPLAY ELAPSED-TIME-OUT
                 ' hours have elapsed between '
                 Vstring-text of START-TIME
                 ' and ' Vstring-text of END-TIME
         ELSE
             DISPLAY 'Error ' Msg-No of FC
                 ' converting ending date to Lilian date'
             STOP RUN
         END-IF
     ELSE
         DISPLAY 'Error ' Msg-No of FC
             ' converting starting date to Lilian date'
         STOP RUN
     END-IF

     GOBACK.
```

Figure 79. Calls to CEESECS in COBOL (Part 2 of 2)

Calls to CEESECS in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMDT1                           */
 /****************************************************/
 /*                                                  */
 /* Function: CEESECS - convert timestamp to seconds */
 /*                                                  */
 /* This example calls the CEESECS callable          */
 /* service to compute the number of hours between   */
 /* the timestamps 11/02/92 05:22 and 11/03/92 17:22.*/
 /* The program responds that 36 hours has elapsed.  */
 /*                                                  */
 /****************************************************/
 CE78DAT : PROC OPTIONS(MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL START_TIME    CHAR(255) VARYING
                      INIT ( '11/02/92 05:22' );
    DCL END_TIME      CHAR(255) VARYING
                      INIT ( '11/03/92 17:22' );
    DCL PICSTR        CHAR(255) VARYING
                      INIT ( 'MM/DD/YY HH:MI' );
    DCL START_SECS    FLOAT8;
    DCL END_SECS      FLOAT8;
    DCL ELAPSED_TIME  FIXED DEC (9,4);
    DCL 01 FC         FEEDBACK;

    /*************************************************/
    /* CEESECS takes the timestamp START_TIME and    */
    /* returns a double-precision Lilian seconds     */
    /* tally in START_SECS.                          */
    /*************************************************/
    CALL CEESECS ( START_TIME, PICSTR, START_SECS, FC );
    IF  FBCHECK( FC, CEE000)  THEN  DO;
       /**********************************************/
       /* CEESECS takes the timestamp END_TIME and   */
       /* returns a double-precision Lilian seconds  */
       /* tally in END_SECS.                         */
       /**********************************************/
       CALL CEESECS ( END_TIME, PICSTR, END_SECS, FC );
       IF  FBCHECK( FC, CEE000)  THEN  DO;
         ELAPSED_TIME = (END_SECS - START_SECS) / 3600;
         PUT SKIP EDIT( ELAPSED_TIME,
                ' hours have elapsed between ',
                START_TIME, ' and ', END_TIME)
             ( F(7,2), (4) A );
         END;
       ELSE  DO;
         PUT SKIP LIST( 'ERROR ' || FC.MsgNo ||
         ' CONVERTING ENDING TIMESTAMP TO SECONDS' );
         STOP;
         END;
       END;
    ELSE DO;
       PUT SKIP LIST( 'ERROR ' || FC.MsgNo
          || ' CONVERTING STARTING TIMESTAMP TO SECONDS' );
       STOP;
       END;

 END CE78DAT ;
```

Figure 80. Calls to CEESECS in PL/I

Examples Illustrating Calls to CEESECS and CEEDATM

The following examples illustrate calls to date and time services to convert a timestamp to seconds (CEESECS), twenty-four hours in seconds is subtracted from the original timestamp value, and a new timestamp is built (CEEDATM) for the updated number of seconds.

Calls to CEESECS and CEEDATM in C

```
/*Module/File Name:  EDCDT2   */
 /*********************************************************/
 /*                                                       */
 /*Function       : CEESECS - convert timestamp to seconds */
 /*               : CEEDATM - convert seconds to timestamp */
 /*               :                                       */
 /*CEESECS is used to convert a timestamp to seconds.      */
 /*24 hours in seconds is subtracted from                 */
 /*the number of seconds in the original timestamp.       */
 /*CEEDATM is then used to build a new timestamp          */
 /*representing the new date and time, 11/01/92 05:22.     */
 /*                                                       */
 /*********************************************************/
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>
#define TimeStamp "11/02/92 05:22"
#define displacement 24
main ()
{
  int User_Input();
  _VSTRING Time_Stamp;
  _CHAR80 New_TimeStamp;
  _VSTRING  picstr;
  _FLOAT8 Lilian_Seconds;
  _FLOAT8 New_Secs;
  _FEEDBACK FC;
  _INT4 dest=2;
  char  New_Time[15];

  /************************************************************
   The date picstr must be set to match the timestamp format.
   ************************************************************/
   strncpy (picstr.string,"MM/DD/YY HH:MI",14);
   picstr.length = 14;

  /************************************************************/
  /* In the following loop the timestamp is converted to Lilian*/
  /* seconds. 24 hours in seconds are subtracted from the      */
  /* Lilian seconds and a new timestamp is created and         */
  /* displayed.                                                */
  /************************************************************/
   strncpy(Time_Stamp.string,TimeStamp,14);
   Time_Stamp.length = 14;
```

Figure 81. Calls to CEESECS and CEEDATM in C (Part 1 of 2)

```
/***************************************************************
 CEESECS takes the user-entered timestamp Time_Stamp and
 returns a double-precision Lilian seconds tally in
 Lilian_Seconds
 ***************************************************************/
 CEESECS ( &Time_Stamp, &picstr , &Lilian_Seconds , &FC );
 if ( (_FBCHECK (FC , CEE000)) == 0 )
 {
/***************************************************************
 The displacement variable is subtracted from the Lilian
 seconds tally in Lilian_Seconds
 ***************************************************************/
   New_Secs = Lilian_Seconds -  displacement * 3600.0;
/**************************************************************
 CEEDATM is invoked to get a new timestamp value based on the
 new Lilian seconds tally in New_Secs.
 **************************************************************/
   CEEDATM ( &New_Secs, &picstr , New_TimeStamp , &FC );
   if ( (_FBCHECK (FC , CEE000)) == 0 )
   {
     New_TimeStamp[14] = '\0';
     sprintf(New_Time,"%s\0",New_TimeStamp);
     printf("%s is the time %i hours before %s\n",
             New_Time, displacement, TimeStamp);
   }
   else
   {
     printf ( "Error converting Seconds to TimeStamp.\n" );
     CEEMSG(&FC, &dest, NULL);
   }
 }
 else
 {
   printf ( "Error converting TimeStamp to seconds.\n" );
   CEEMSG(&FC, &dest, NULL);
 }
}
```

Figure 81. Calls to CEESECS and CEEDATM in C (Part 2 of 2)

Calls to CEESECS and CEEDATM in COBOL

```
CBL LIB,APOST
      *Module/File Name: IGZTDT2
      *************************************************************
      **                                                         **
      ** CEE80DAT - Call CEESECS to convert timestamp to seconds**
      **           and CEEDATM to convert seconds to timestamp **
      **                                                         **
      **  CEESECS is used to convert a timestamp to seconds.    **
      **  24 hours in seconds is subtracted from                **
      **  the number of seconds in the original timestamp.      **
      **  CEEDATM is then used to build a new timestamp for     **
      **  the updated number of seconds.                        **
      **                                                         **
      *************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CE80DAT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      *  Double precision needed for the seconds results
       01  START-SECS             COMP-2.
       01  NEW-TIME               COMP-2.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity       PIC S9(4) BINARY.
                   04  Msg-No         PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code     PIC S9(4) BINARY.
                   04  Cause-Code     PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info       PIC S9(9) BINARY.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char       PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of PICSTR.
       01  WS-TIMESTAMP.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char       PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of WS-TIMESTAMP.
       01  NEW-TIMESTAMP          PIC X(80).
       01  INPUT-VARIABLES.
           05  SECONDS-DISPLACED   PIC S9(9) BINARY.
           05  ELAPSED-TIME-OUT    PIC +Z(4)9.99.

       PROCEDURE DIVISION.
```

Figure 82. Calls to CEESECS and CEEDATM in COBOL (Part 1 of 2)

```
 0001-BEGIN-PROCESSING.
     MOVE  14 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY HH:MI' TO Vstring-text of PICSTR.
     MOVE 14 TO Vstring-length of WS-TIMESTAMP.
     MOVE '11/02/92 05:22' TO Vstring-text of WS-TIMESTAMP.
*    **********************************************************
*    * CEESECS is invoked to obtain the Lilian seconds tally  *
*    * corresponding to the timestamp 11/02/92 05:22.         *
*    * The Lilian seconds tally is returned in the double-    *
*    * precision floating-point field START-SECS.             *
*    **********************************************************
     CALL 'CEESECS' USING WS-TIMESTAMP, PICSTR, START-SECS, FC.
     IF CEE000 of FC THEN
*        **********************************************
*        * The Lilian seconds tally in START-SECS is  *
*        * decremented by 24 hours worth of seconds.. *
*        **********************************************
         COMPUTE NEW-TIME = START-SECS - 24 * 3600
*        **************************************************
*        * CEEDATM is invoked to obtain a new timestamp   *
*        * based on the new Lilian seconds tally.         *
*        **************************************************
         CALL 'CEEDATM' USING NEW-TIME, PICSTR, NEW-TIMESTAMP, FC
         IF CEE000 of FC THEN
             DISPLAY 'The time  24 hours before '
                 Vstring-text of WS-TIMESTAMP
                 ' is ' NEW-TIMESTAMP
         ELSE
             DISPLAY 'Error converting seconds to timestamp.'
             STOP RUN
         END-IF
     ELSE
         DISPLAY 'Error converting timestamp to seconds.'
         STOP RUN
     END-IF

     GOBACK.
```

Figure 82. Calls to CEESECS and CEEDATM in COBOL (Part 2 of 2)

Calls to CEESECS and CEEDATM in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMDT2                            */
 /****************************************************/
 /*                                                  */
 /* Function: CEESECS - convert timestamp to seconds */
 /*         : CEEDATM - convert seconds to timestamp */
 /*                                                  */
 /* CEESECS is used to convert a timestamp to        */
 /* seconds. 24 hours in seconds is subtracted from  */
 /* the number of seconds in the original timestamp. */
 /* CEEDATM is then used to build a new timestamp    */
 /* representing the new date and time.              */
 /*                                                  */
 /****************************************************/
 PLIDS: PROC OPTIONS(MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL TIMESTAMP      CHAR(255) VARYING
                       INIT('01/26/67 20:00');
    DCL NEW_TIMESTAMP  CHAR(80);
    DCL PICSTR         CHAR(255) VARYING
                       INIT ( 'MM/DD/YY HH:MI' );
    DCL START_SECS     FLOAT8;
    DCL NEW_TIME       FLOAT8;
    DCL DISPLACEMENT   INT4 INIT(24);
    DCL 01 FC          FEEDBACK;
    /************************************************/
    /* CEESECS is invoked to obtain the Lilian      */
    /* seconds tally corresponding to the timestamp  */
    /* 01/26/67 20:00.  The Lilian seconds tally is  */
    /* returned in double-precision variable        */
    /* START_SECS.                                  */
    /************************************************/
    CALL CEESECS ( TIMESTAMP, PICSTR, START_SECS, FC );
    IF  FBCHECK( FC, CEE000)  THEN  DO;
       /*********************************************/
       /* The Lilian seconds tally in START_SECS is  */
       /* decremented by 24 hour DISPLACEMENT       */
       /* variable times 3600 seconds.              */
       /******************************************** */
       NEW_TIME = START_SECS - DISPLACEMENT * 3600;
       /*********************************************/
       /* CEEDATM is invoked to obtain a new        */
       /* TimeStamp based on the new Lilian seconds. */
       /*********************************************/
       CALL CEEDATM ( NEW_TIME, PICSTR, NEW_TIMESTAMP, FC );
       IF  FBCHECK( FC, CEE000)  THEN  DO;
          PUT SKIP LIST ( 'The time ' || DISPLACEMENT
             || ' hours before ' || TIMESTAMP
             || ' is ' || NEW_TIMESTAMP );
          END;
       ELSE  DO;
          PUT SKIP LIST('ERROR CONVERTING SECONDS TO TIMESTAMP');
          PUT SKIP LIST( 'CEEDATM failed with msg '|| FC.MsgNo );
          END;
       END;
    ELSE  DO;
       PUT SKIP LIST('ERROR CONVERTING TIMESTAMP TO SECONDS' );
       PUT SKIP LIST( 'CEESECS failed with msg '|| FC.MsgNo );
       END;

 END PLIDS;
```

Figure 83. Calls to CEESECS and CEEDATM in PL/I

Examples Illustrating Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM

The following examples illustrate calls to date and time services to convert a timestamp into seconds (CEESECS), convert the seconds to a date and time component (CEESECI), add thirty-two months to the month component, convert the date and time component back to seconds (CEEISEC), and build a new timestamp (CEEDATM).

Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in C

```
 /*Module/File Name: EDCDT3  */
 /*******************************************************************/
 /*Function      : CEESECS - convert timestamp to seconds           */
 /*              : CEESECI - convert seconds to time components     */
 /*              : CEEISEC - convert time components to seconds     */
 /*              : CEEDATM - convert seconds to timeStamp           */
 /*              :                                                  */
 /*32 months is added to the timestamp 11/02/92 05:22 giving        */
 /*the new timestamp 07/02/95 05:22.                                */
 /*                                                                 */
 /*CEESECS is used to convert timestamp 11/02/92 05:22 to seconds.  */
 /*CEESECI is used to convert the seconds to date/time components.  */
 /*32 months is added to the month component.                       */
 /*CEEISEC is then used to convert date/time components to seconds. */
 /*CEEDATM is then used to build a new timestamp for the new time.  */
 /*******************************************************************/
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <ceeedcct.h>
#define TimeStamp "11/02/92 05:22"
#define displacement 32
void main ()
{
  _VSTRING Time_Stamp;
  _CHAR80 New_TimeStamp;
  _VSTRING  picstr;
  _FLOAT8 Lilian_Seconds;
  _FLOAT8 New_Secs;
  _FEEDBACK FC;
   char  New_Time[15];

  int Month_in_Century;
  /******************************************
    Date/time components for CEESECI, CEEISEC.
   ******************************************/
   _INT4 year;
   _INT4 month;
   _INT4 days;
   _INT4 hours;
   _INT4 minutes;
   _INT4 seconds;
   _INT4 millsec;

  /*************************************************************
   The date picstr must be set to match the timestamp format.
   *************************************************************/
   strcpy (picstr.string,"MM/DD/YY HH:MI");
   picstr.length = 14;
   strncpy(Time_Stamp.string,TimeStamp,14);
   Time_Stamp.length = 14;
```

Figure 84. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in C (Part 1 of 2)

```
  /*******************************************************
   CEESECS takes the timestamp "11/02/92 05:22" and returns
   a double-precision Lilian seconds tally in Lilian_Seconds
   *******************************************************/
   CEESECS ( &Time_Stamp, &picstr , &Lilian_Seconds , &FC );
   if ((_FBCHECK (FC, CEE000)) == 0)
    {
  /*************************************************************
   CEESECI converts the Lilian seconds tally in Lilian_Seconds and
   returns date/time components.
   *************************************************************/
      CEESECI ( &Lilian_Seconds, &year, &month, &days, &hour
                &minutes, &seconds, &millsec, &FC);
      if ((_FBCHECK (FC, CEE000)) == 0)
       {
  /*************************************************************
   The month component of the timestamp is converted to
   month-in-century.
   Then a new month and a new year are computed from the
   new month-in-century number. The month date/time component has a
   range between 1 and 12.
   *************************************************************/
         Month_in_Century = year*12 + month + displacement - 1;
         year = Month_in_Century / 12;
         month = (Month_in_Century % 12) + 1;
  /* ************************************************************
   The month date/time component has been shifted
   forward 32 months. Our examples gets a new Lilian seconds
   tally based on the new month and year components.
   This is done with a call to function CEEISEC.
   The new Lilian seconds tally is placed in the double-precision
   variable Lilian_Seconds.
   *************************************************************/
         CEEISEC (&year,
                  &month,
                  &days,
                  &hours,
                  &minutes,
                  &seconds,
                  &millsec, &Lilian_Seconds, &FC );
         if ((_FBCHECK (FC, CEE000)) == 0)
          {
  /**********************************************************
   CEEDATM is invoked to get a new timestamp value based on the
   new Lilian seconds tally in Lilian_Seconds.
   **********************************************************/
            CEEDATM ( &Lilian_Seconds,
                      &picstr ,
                      New_TimeStamp ,
                      &FC );
            if ((_FBCHECK (FC, CEE000)) == 0)
             {
               New_TimeStamp[14] = '\0';
               sprintf(New_Time,"%s\0",New_TimeStamp);
               if ( displacement < 0 )
                printf("%s is the time %d months before %s.\n",
                          New_Time, displacement, TimeStamp);
               else
                printf("%s will be the time %d months after %s.\n",
                           New_Time, displacement, TimeStamp);
             }
            else
               printf ( "Error converting Seconds to TimeStamp.\n" );
          }
         else
           printf ( "Error converting Components to seconds.\n" );
       }
     else
        printf ( "Error converting seconds to components.\n" );
    }
  else
   printf ( "Error converting TimeStamp to seconds\n" );
}
```

Figure 84. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in C (Part 2 of 2)

Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in COBOL

```
CBL LIB,APOST
      *Module/File Name: IGZTDT3
      ****************************************************************
      *                                                              *
      * CE81DATA - Call the following LE service routines:          *
      *           : CEESECS - convert timestamp to seconds           *
      *           : CEESECI - convert seconds to time components     *
      *           : CEEISEC - convert time components to seconds     *
      *           : CEEDATM - convert seconds to timestamp           *
      *                                                              *
      *   CEESECS is used to convert the timestamp to seconds        *
      *   CEESECI is used to convert seconds to date/time components.*
      *   32 months is added to the month and year component         *
      *       of date/time.                                          *
      *   CEEISEC is to convert the date/time components with the    *
      *       new months component back to a Lilian seconds tally.   *
      *   CEEDATM is then used to build a new timestamp for          *
      *       the updated number of seconds.                         *
      *                                                              *
      ****************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CE81DAT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      *  Double precision needed for the seconds results
       01  START-SECS              COMP-2.
       01  NEW-TIME                COMP-2.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity        PIC S9(4) BINARY.
                   04  Msg-No          PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code      PIC S9(4) BINARY.
                   04  Cause-Code      PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info        PIC S9(9) BINARY.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of PICSTR.
       01  WS-TIMESTAMP.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of WS-TIMESTAMP.
       01  NEW-TIMESTAMP           PIC X(80).
      *    ********************************************
      *    * These are the date/time variables used by *
      *    * CEEISEC and CEESECI.                      *
      *    ********************************************
```

Figure 85. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in COBOL (Part 1 of 3)

```
       01  DATE-TIME-COMPONENTS BINARY.
           05  YEAR               PIC 9(9).
           05  MONTH              PIC 9(9).
           05  DAYS               PIC 9(9).
           05  HOURS              PIC 9(9).
           05  MINUTES            PIC 9(9).
           05  SECONDS            PIC 9(9).
           05  MILLSEC            PIC 9(9).
       01  FILLER                 PIC X(80).
       01  INPUT-VARIABLES.
           05  MONTHS-TO-DISPLACE  PIC S9(4) BINARY VALUE 32.
           05  DISPLACEMENT-COMP   PIC S9(4) BINARY.
           05  MONTHNUM            PIC 9(9)  BINARY.

       PROCEDURE DIVISION.

       0001-BEGIN-PROCESSING.
           MOVE 14 TO Vstring-length of WS-TIMESTAMP.
           MOVE '11/02/92 05:22' TO Vstring-text of WS-TIMESTAMP.
           MOVE 14 TO Vstring-length of PICSTR.
           MOVE 'MM/DD/YY HH:MI' TO Vstring-text of PICSTR.
      *    ***********************************************************
      *    * The timestamp '11/02/92 05:22' is converted to          *
      *    * seconds under the control of the mask PICSTR.  CEESECS  *
      *    * will return a Lilian seconds tally in the double-       *
      *    * precision floating-point variable START-SECS.           *
      *    ***********************************************************
           CALL 'CEESECS' USING WS-TIMESTAMP, PICSTR, START-SECS, FC.
           IF CEE000 of FC THEN
      *    ***********************************************************
      *    * The Lilian seconds tally in field START-SECS is mapped  *
      *    * into its date/time components using function CEESECI.  *
      *    ***********************************************************
               CALL 'CEESECI' USING START-SECS, YEAR, MONTH, DAYS,
                                    HOURS, MINUTES, SECONDS, MILLSEC, FC
               IF CEE000 of FC THEN
                   MOVE MONTHS-TO-DISPLACE TO DISPLACEMENT-COMP

      *            ***************************************************
      *            * MONTH is converted to month-in-century for the  *
      *            * displacement arithmetic. Then a new month and   *
      *            * year are computed from the new month-in-century *
      *            * number (in variable MONTHNUM).  The months com- *
      *            * ponent has an allowed range of between 1 and 12. *
      *            ***************************************************
                   COMPUTE MONTHNUM =
                       YEAR * 12 + MONTH + DISPLACEMENT-COMP - 1
                   DIVIDE MONTHNUM BY 12 GIVING YEAR REMAINDER MONTH
                   ADD 1 TO MONTH
      *            **************************************************
      *            * Now that the MONTH DateTime component has      *
      *            * been shifted forward by 32 months,             *
      *            * we must get a new Lilian seconds tally based   *
      *            * on the new MONTH and YEAR components.  We      *
      *            * do this with a call to the CEEISEC callable    *
      *            * service.  The new Lilian seconds tally is      *
      *            * placed in the double-precision field NEW-TIME. *
      *            **************************************************
                   CALL 'CEEISEC' USING YEAR, MONTH, DAYS, HOURS,
                               MINUTES, SECONDS, MILLSEC, NEW-TIME, FC
      *            ******************************************
      *            * CEEDATM is now used to obtain a new    *
      *            * timestamp based on the Lilian seconds  *
      *            * tally in the variable New-time.        *
      *            ******************************************
```

Figure 85. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in COBOL (Part 2 of 3)

```
        IF CEE000 THEN
            CALL 'CEEDATM' USING NEW-TIME, PICSTR,
                                 NEW-TIMESTAMP, FC
            IF CEE000 THEN
                DISPLAY 'The time '
                    MONTHS-TO-DISPLACE ' months after '
                    Vstring-text of WS-TIMESTAMP
                    ' is ' NEW-TIMESTAMP
            ELSE
                DISPLAY 'Error ' Msg-No of FC
                    ' converting seconds to timestamp.'
            END-IF
        ELSE
            DISPLAY 'Error ' Msg-No of FC
                ' converting components to seconds.'
        END-IF
    ELSE
        DISPLAY 'Error ' Msg-No of FC
            ' converting seconds to components.'
    END-IF
ELSE
    DISPLAY 'Error ' Msg-No of FC
        ' converting timestamp to seconds.'
END-IF

GOBACK.
```

Figure 85. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in COBOL (Part 3 of 3)

Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMDT3
 /****************************************************************/
 /*                                                              */
 /* Function      : CEESECS - convert timestamp to seconds       */
 /*               : CEESECI - convert seconds to time components  */
 /*               : CEEISEC - convert time components to seconds  */
 /*               : CEEDATM - convert seconds to timestamp       */
 /*               :                                              */
 /* 32 months is added to the timestamp 11/02/92 05:22           */
 /*    giving the new timestamp 07/02/95 05:22.                  */
 /*                                                              */
 /* CEESECS is used to convert the timestamp to seconds          */
 /* CEESECI is used to convert seconds to date/time components    */
 /* 32 months is added to the month component.                   */
 /* CEEISEC is used to convert the date components to seconds.    */
 /* CEEDATM is then used to build a new timestamp for the         */
 /*    updated time.                                             */
 /*                                                              */
 /****************************************************************/
 CE81DAT: PROC OPTIONS(MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL TIMESTAMP       CHAR(255) VARYING INIT( '11/02/92 05:22' );
    DCL NEW_TIMESTAMP   CHAR(80);
    DCL PICSTR          CHAR(255) VARYING INIT( 'MM/DD/YY HH:MI' );
    DCL START_SECS      FLOAT8;
    DCL NEW_TIME        FLOAT8;
    DCL DISPLACEMENT    INT4 INIT(32);
    DCL MONTHNUM        INT4;
    DCL 01 FC           FEEDBACK;
    /****************************************/
    /* DATE COMPONENTS FOR CEESECI, CEEISEC   */
    /****************************************/
    DCL YEAR            INT4;
    DCL MONTH           INT4;
    DCL DAYS            INT4;
    DCL HOURS           INT4;
    DCL MINUTES         INT4;
    DCL SECONDS         INT4;
    DCL MILLSEC         INT4;

    /*******************************************************/
    /* The timestamp '11/02/92 05:22' is converted to seconds */
    /* under the control of the mask PICSTR. CEESECS will     */
    /* return a Lilian seconds tally in the double-precision  */
    /* floating-point field START_SECS.                      */
    /*******************************************************/
    CALL CEESECS ( TIMESTAMP, PICSTR, START_SECS, FC );
    IF  FBCHECK( FC, CEE000)  THEN  DO;
    /*************************************************************/
    /* The Lilian seconds tally in the field START_SECS is mapped  */
    /* into its date/time components using function CEESECI.      */
    /*************************************************************/
       CALL CEESECI( START_SECS, YEAR, MONTH, DAYS, HOURS, MINUTES,
          SECONDS, MILLSEC, FC);
```

Figure 86. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in PL/I (Part 1 of 2)

```
      IF  FBCHECK( FC, CEE000)  THEN  DO;
         /************************************************************/
         /* MONTH is converted to month-in-century for the displace- */
         /* ment arithmetic. Then a new month and year are computed  */
         /* from the new month-in-century number. The months         */
         /* component has an allowed range of between 1 and 12.      */
         /************************************************************/
         MONTHNUM = YEAR * 12 + MONTH + DISPLACEMENT - 1;
         YEAR = MONTHNUM / 12;
         MONTH = MOD(MONTHNUM, 12) + 1;
         /***********************************************************/
         /* Now that the MONTH DateTime component has been shifted  */
         /* forward by 32 months, we must get a new Lilian          */
         /* seconds tally based on the new MONTH and YEAR compo-    */
         /* nents.  We do this with a call to service CEEISEC.      */
         /* The new Lilian seconds tally is placed in the double-   */
         /* precision floating- point variable NEW_TIME.            */
         /***********************************************************/
         CALL CEEISEC (YEAR, MONTH, DAYS, HOURS, MINUTES, SECONDS,
            MILLSEC, NEW_TIME, FC);
         IF  FBCHECK( FC, CEE000)  THEN  DO;
            /*******************************************************/
            /* CEEDATM is now used to obtain a new timestamp based  */
            /* on the Lilian seconds tally in variable New_time     */
            /*******************************************************/
            CALL CEEDATM( NEW_TIME, PICSTR, NEW_TIMESTAMP, FC );
            IF  FBCHECK( FC, CEE000) THEN DO;
               PUT SKIP EDIT( 'The time ', DISPLACEMENT,
                     ' months after ', TIMESTAMP,
                     ' is ', NEW_TIMESTAMP  )
                  (A, F(4), (3) A);
                END;
             ELSE  DO;
                PUT SKIP EDIT( 'ERROR ', FC.MsgNo,
                      ' CONVERTING SECONDS TO TIMESTAMP')
                   (A, F(4), A);
                END;
             END;
          ELSE  DO;
             PUT SKIP EDIT( 'ERROR ', FC.MsgNo,
                   ' CONVERTING COMPONENTS TO SECONDS')
                (A, F(4), A);
             END;
          END;
       ELSE  DO;
          PUT SKIP EDIT( 'ERROR ', FC.MsgNo,
                ' CONVERTING SECONDS TO COMPONENTS' )
             (A, F(4), A);
          END;
       END;
   ELSE  DO;
       PUT SKIP EDIT( 'ERROR ', FC.MsgNo,
             ' CONVERTING TIMESTAMP TO SECONDS' )
          (A, F(4), A);
       END;
END CE81DAT;
```

Figure 86. Calls to CEESECS, CEESECI, CEEISEC, and CEEDATM in PL/I (Part 2 of 2)

Example Illustrating Calls to CEEDAYS, CEEDATE, and CEEDYWK

The following examples illustrate calls to date and time services to convert a date to a Lilian date (CEEDAYS). In these examples, a varying number of days are added to the Lilian date, the date is converted back to a character format (CEEDATE), and the day of the week for that Lilian date is returned (CEEDYWK).

Calls to CEEDAYS, CEEDATE, and CEEDYWK in C

```
/*Module/File Name:  EDCDT4   */
 /*********************************************************/
 /*Function       : CEEDAYS - convert date to Lilian date  */
 /*               : CEEDATE - convert Lilian date to date  */
 /*               : CEEDYWK - find day-of-week from Lilian */
 /*               :                                        */
 /*CEEDAYS is passed the calendar date "11/09/92". The date*/
 /*is originally in YYMMD format and conversion to Lilian  */
 /*format takes place. On return, a varying number of days */
 /*is added to or subtracted from the Lilian date.         */
 /*CEEDATE is called to convert the Lilian dates to the    */
 /*calendar format "MM/DD/YY".                             */
 /*CEEDYWK is called to return the day of the week for     */
 /*each derived Lilian date.                               */
 /*The results are tested for accuracy.                    */
 /*********************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <leawi.h>
#include <ceeedcct.h>
  char PastFuture;
  int NumberOfDays[5] = { 80, 20, 10, 5, 4};
  int i;
void main ()
{
  _CHAR80 chrdate;
  _VSTRING  picstr;
  _VSTRING CurrentDate;
  _INT4 Current_Lilian;
  _INT4 Displaced_Lilian;
  _INT4 WeekDay;
  _INT4 ChkWeekDay[5] = { 6, 1, 5, 4, 6 };
  _FEEDBACK FC;
  char Entered_Date[8];
  _INT4 dest=2;

  struct tm *timeptr;
  char   Current_Date[6];
  time_t current_time;
  char *ChkDates[] = {
                "08/21/92",
                "11/29/92",
                "11/19/92",
                "11/04/92",
                "11/13/92",
               };
  /**********************************************************/
  /* Set current date to 11/09/92 in YYMMDD format          */
  /**********************************************************/
   strncpy (CurrentDate.string,"921109",6);
   CurrentDate.length = 6;
 /***************************************************************/
  /* The date picstr must be adjusted to fit the current date    */
  /* format.                                                     */
  /***************************************************************/
   strncpy (picstr.string,"YYMMDD",6);
   picstr.length = 6;
```

Figure 87. Calls to CEEDAYS, CEEDATE, and CEEDYWK in C (Part 1 of 2)

```
 /**************************************************************/
 /*Call CEEDAYS to convert the date in Current_Date to its     */
 /*corresponding Lilian date format.                           */
 /**************************************************************/
  CEEDAYS ( &CurrentDate, &picstr , &Current_Lilian , &FC );
  if ( _FBCHECK (FC , CEE000) != 0 )
    {
     printf ("Error in converting current date.\n");
     CEEMSG(&FC, &dest, NULL);
     exit(99);
    }

 /**************************************************************/
 /*Modify the date picstr to the familiar MM/DD/YY format.     */
 /**************************************************************/
  strncpy (picstr.string,"MM/DD/YY",8);
  picstr.length = 8;
 /*************************************************************/
 /* In the following loop, add or subtract the number         */
 /* of days in each element of the NumberOfDays array to the  */
 /* Lilian date.  Determine the day of the week for each      */
 /* Lilian date and convert each date back to "MM/DD/YY"      */
 /* format. Issue a message if anything goes wrong.           */
 /*************************************************************/
  for (i=0; i < 5; i++)
   {
    if (i == 0 || i == 3)
       Displaced_Lilian = Current_Lilian - NumberOfDays[i];
    else
       Displaced_Lilian = Current_Lilian + NumberOfDays[i];
  /************************************************************/
  /*Call CEEDATE to convert the Lilian dates to MM/DD/YY      */
  /*format.                                                   */
  /************************************************************/
    CEEDATE ( &Displaced_Lilian, &picstr , chrdate , &FC );
    if ( _FBCHECK (FC , CEE000) == 0 )
    {
       chrdate[8] = '\0';
  /************************************************************/
  /*Compare the dates to an array of expected values.         */
  /*Issue an error message if any conversion is incorrect.    */
  /************************************************************/
       if ( memcmp ( &chrdate, ChkDates[i] , 8) != 0)
         printf (
          "Error  in returned date %8s for displacement %d\n",
             chrdate,NumberOfDays[i]);
  /****************************************************************/
  /*Call CEEDYWK to return the day-of-the-week value (1 thru 7)   */
  /*for each calculated Lilian date. Compare results to an array  */
  /*of expected returned values and issue an error message for any*/
  /*incorrect values.                                             */
  /****************************************************************/
       CEEDYWK ( &Displaced_Lilian , &WeekDay , &FC );
       if ( _FBCHECK (FC , CEE000) == 0 )
       {
         if ( WeekDay != ChkWeekDay[i])
          printf ( "Error in day of the week for %s\n",
                   chrdate);
       }
       else
       {
         printf ("Error finding day of the week\n");
         CEEMSG(&FC, &dest, NULL);
       }
    }
    else
    {
      printf ( "Error converting Lilian date to date.\n" );
      CEEMSG(&FC, &dest, NULL);
    }
   } /* for loop */
}
```

Figure 87. Calls to CEEDAYS, CEEDATE, and CEEDYWK in C (Part 2 of 2)

Calls to CEEDAYS, CEEDATE, and CEEDYWK in COBOL

```
CBL LIB,APOST
      *Module/File Name: IGZTDT4
      ************************************************************
      **                                                        **
      ** CE77DAT - Call the following LE service routines:      **
      **         : CEEDAYS - convert date to Lilian format      **
      **         : CEEDATE - convert Lilian date to date        **
      **         : CEEDYWK - find day of week from Lilian       **
      **                                                        **
      ** CEEDAYS is passed the calendar date '11/09/92'. The    **
      ** date is originally in YYMMDD format and conversion to  **
      ** Lilian format takes place. On return from CEEDAYS,     **
      ** a varying  number of days is added to or subtracted    **
      ** from the Lilian date.                                  **
      ** CEEDATE is then called to convert the Lilian dates to  **
      **     the format 'MM/DD/YY'.                             **
      ** CEEDYWK is called to return the day of the week for    **
      **     each derived Lilian date.                          **
      ** The results are tested for accuracy.                   **
      **                                                        **
      ************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CE77DAT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  WEEKDAY                PIC S9(9) BINARY.
       01  LILIAN                 PIC S9(9) BINARY.
       01  CURRENT-LILIAN         PIC S9(9) BINARY.
       01  DISPLACED-LILIAN       PIC S9(9) BINARY.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity       PIC S9(4) BINARY.
                   04  Msg-No         PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code     PIC S9(4) BINARY.
                   04  Cause-Code     PIC S9(4) BINARY.
               03  Case-Sev-Ctl   PIC X.
               03  Facility-ID    PIC XXX.
           02  I-S-Info       PIC S9(9) BINARY.
       01  INDXX                  PIC S9(9) BINARY.
       01  NUMBER-OF-DAYS.
           05  NUMBERS.
               10  FILLER         PIC S9(9) BINARY VALUE 80.
               10  FILLER         PIC S9(9) BINARY VALUE 20.
               10  FILLER         PIC S9(9) BINARY VALUE 10.
               10  FILLER         PIC S9(9) BINARY VALUE  5.
               10  FILLER         PIC S9(9) BINARY VALUE  4.
           05  NUMBEROFDAYS REDEFINES NUMBERS
                                  PIC S9(9) BINARY OCCURS 5 TIMES.
       01  PICSTR.
           02  Vstring-length     PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char       PIC X,
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of PICSTR.
       01  CHRDATE             PIC X(80).
```

Figure 88. Calls to CEEDAYS, CEEDATE, and CEEDYWK in COBOL (Part 1 of 3)

```
 01  CURRENT-DATE.
     02  Vstring-length      PIC S9(4) BINARY.
     02  Vstring-text.
         03  Vstring-char        PIC X
                     OCCURS 0 TO 256 TIMES
                     DEPENDING ON Vstring-length
                        of CURRENT-DATE.
 01  INPUT-VARIABLES.
     05  DATE-TABLE.
        10  FILLER           PIC X(9) VALUE '08/21/92'.
        10  FILLER           PIC X(9) VALUE '11/29/92'.
        10  FILLER           PIC X(9) VALUE '11/19/92'.
        10  FILLER           PIC X(9) VALUE '11/04/92'.
        10  FILLER           PIC X(9) VALUE '11/13/92'.
     05  CHKDATES REDEFINES DATE-TABLE PIC X(9)
            OCCURS 5 TIMES.
 01  CHK-WEEKDAYS.
     05 DAY-TABLE.
        10  FILLER           PIC S9(9) BINARY VALUE 6.
        10  FILLER           PIC S9(9) BINARY VALUE 1.
        10  FILLER           PIC S9(9) BINARY VALUE 5.
        10  FILLER           PIC S9(9) BINARY VALUE 4.
        10  FILLER           PIC S9(9) BINARY VALUE 6.
     05 CHKWEEKDAY REDEFINES DAY-TABLE PIC S9(9) BINARY
            OCCURS 5 TIMES.

 PROCEDURE DIVISION.

 0001-BEGIN-PROCESSING.
     DISPLAY '*** Example CE77DAT in motion'
*    *********************************************************
*    * The current date is converted to a Lilian date.      *
*    *********************************************************
     MOVE  6 TO Vstring-length of PICSTR.
     MOVE 'YYMMDD' TO Vstring-text of PICSTR.
     MOVE  6 TO Vstring-length of CURRENT-DATE.
     MOVE '921109' TO Vstring-text of CURRENT-DATE.
*    *********************************************************
*    * Call CEEDAYS to return the Lilian days tally for the  *
*    * date value in the variable CURRENT-DATE.              *
*    *********************************************************
     CALL 'CEEDAYS' USING CURRENT-DATE, PICSTR,
                          CURRENT-LILIAN, FC.
     IF NOT CEE000 THEN
         DISPLAY 'Error ' Msg-No of FC
              ' in converting current date'
     END-IF.
*    *********************************************************
*    * The datestamp mask must be changed for the dates      *
*    * being entered by the user.                            *
*    *********************************************************
     MOVE  8 TO Vstring-length of PICSTR.
     MOVE 'MM/DD/YY' TO Vstring-text of PICSTR.
```

Figure 88. Calls to CEEDAYS, CEEDATE, and CEEDYWK in COBOL (Part 2 of 3)

```
*    ***********************************************************
*    * In the following loop, add or subtract the number of    *
*    * days in each element of the NumberofDays array to the   *
*    * Lilian date. Determine the day of the week for each     *
*    * Lilian date and convert each date back to 'MM/DD/YY'    *
*    * format. Issue a message if anything goes wrong.         *
*    ***********************************************************
     MOVE 1 TO INDXX.
     PERFORM UNTIL INDXX = 6
         IF (INDXX = 1 OR 4) THEN
             COMPUTE DISPLACED-LILIAN =
                 CURRENT-LILIAN - NUMBEROFDAYS(INDXX)
         ELSE
             COMPUTE DISPLACED-LILIAN =
                 CURRENT-LILIAN + NUMBEROFDAYS(INDXX)
         END-IF
*        ****************************************************
*        * Call CEEDATE to convert the Lilian dates to      *
*        * MM/DD/YY format.                                 *
*        ****************************************************
         CALL 'CEEDATE' USING DISPLACED-LILIAN, PICSTR,
                          CHRDATE, FC
         IF CEE000 THEN
*            ************************************************
*            * Compare converted date to expected value     *
*            ************************************************
             IF CHRDATE NOT = CHKDATES(INDXX) THEN
                 DISPLAY 'Expecting returned date of '
                     CHKDATES(INDXX)
                     ' for displacement of ' NUMBEROFDAYS(INDXX)
                     ', but got returned date of ' CHRDATE
             END-IF
*            *****************************************************
*            * Call CEEDYWK to return a day-of-the week value (1 *
*            * thru 7) for each calculated Lilian date. Compare  *
*            * results to an array of expected values and issue  *
*            * an error message for any incorrect values.        *
*            *****************************************************
             CALL 'CEEDYWK' USING DISPLACED-LILIAN, WEEKDAY, FC
             IF CEE000 THEN
                 IF  WEEKDAY NOT = CHKWEEKDAY(INDXX) THEN
                     DISPLAY 'Expecting day of week '
                         CHKWEEKDAY(INDXX) ', but got ' WEEKDAY
                         ' instead for ' CHRDATE
                 END-IF
             ELSE
                 DISPLAY 'Error ' Msg-No of FC
                     ' in finding day-of-week'
             END-IF
         ELSE
             DISPLAY 'Error ' Msg-No of FC
                 ' converting date to Lilian date'
         END-IF
         ADD 1 TO INDXX
     END-PERFORM.

     DISPLAY '*** Example CE77DAT complete'

     STOP RUN.
```

Figure 88. Calls to CEEDAYS, CEEDATE, and CEEDYWK in COBOL (Part 3 of 3)

Calls to CEEDAYS, CEEDATE, and CEEDYWK in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMDT4
 /***********************************************************/
 /*                                                         */
 /* Function      : CEEDAYS - convert date to Lilian date   */
 /*               : CEEDATE - convert Lilian Date to date   */
 /*               : CEEDYWK - find day-of-week from Lilian  */
 /*                                                         */
 /* CEEDAYS is passed the calendar date "11/09/92". The     */
 /* date is originally in YYMMD format and conversion to    */
 /* Lilian format takes place. On return, a varying number  */
 /* of days is added to or subtracted from the Lilian date. */
 /* CEEDATE is called to convert the Lilian dates to the    */
 /* calendar format "MM/DD/YY".  CEEDYWK is called to       */
 /* return the day of the week for each derived Lilian date. */
 /*                                                         */
 /* The results are tested for accuracy.                    */
 /*                                                         */
 /***********************************************************/
 CE77DAT: PROC OPTIONS(MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL CHRDATE          CHAR(80);
    DCL CURRENT_DATE     CHAR(255) VARYING;
    DCL PICSTR           CHAR(255) VARYING;
    DCL Lilian           INT4;
    DCL ii               INT4;
    DCL NumberOfDays (5) INT4
                         INIT( 80, 20, 10, 5, 4);
    DCL ChkWeekDay   (5) INT4
                         INIT(  6,  1,  5, 4, 6);
    DCL CURRENT_LILIAN   INT4;
    DCL DISPLACED_LILIAN INT4;
    DCL WEEKDAY          INT4;
    DCL 01 FC            FEEDBACK;
    DCL ChkDates (5)     CHAR(8) INIT(
                            '08/21/92',
                            '11/29/92',
                            '11/19/92',
                            '11/04/92',
                            '11/13/92');

    PUT SKIP LIST( '>>> Example CE77DAT in motion');
    /*********************************************************/
    /* Set current date to 11/09/92 in YYMMDD format         */
    /*********************************************************/
    Picstr = 'YYMMDD';
    Current_Date = '921109';
    /*********************************************************/
    /* Call CEEDAYS to convert the date in Current_Date to   */
    /* its corresponding Lilian date format.                 */
    /*********************************************************/
    Call CEEDAYS ( Current_Date, Picstr, Current_Lilian, FC );
    IF ¬ FBCHECK( FC, CEE000)  THEN  DO;
         PUT SKIP LIST( 'Error in converting Current Date');
         END;
```

Figure 89. Calls to CEEDAYS, CEEDATE, and CEEDYWK in PL/I (Part 1 of 2)

```
   /************************************************************/
   /* The date picstr must be adjusted to fit the current      */
   /* date format.                                             */
   /************************************************************/
   Picstr = 'MM/DD/YY';
   /**************************************************************/
   /* In the following loop, add or subtract the number          */
   /* of days in each element of the NumberOfDays array to the   */
   /* Lilian date.  Determine the day of the week for each       */
   /* Lilian date and convert each date back to "MM/DD/YY"       */
   /* format.  Issue a message if anything goes wrong.           */
   /**************************************************************/
   DO ii = 1 TO 5;
      IF ( ii= 1 | ii= 4 )  THEN  DO;
         Displaced_Lilian = Current_Lilian - NumberOfDays(ii);
         END;
      ELSE  DO;
         Displaced_Lilian = Current_Lilian + NumberOfDays(ii);
         END;
      /***********************************************************/
      /* Call CEEDATE to convert the Lilian dates to MM/DD/YY    */
      /* format.                                                 */
      /***********************************************************/
      Call CEEDATE ( Displaced_Lilian, Picstr, ChrDate, FC );
      IF FBCHECK( FC, CEE000)  THEN  DO;
         /*********************************************************/
         /* Compare the dates to an array of expected values.     */
         /* Issue an error message if any conversion is incorrect. */
         /*********************************************************/
         IF  ChrDate ¬= ChkDates(ii)  THEN  DO;
            PUT SKIP EDIT( 'Error in returned date ', Chrdate,
                  ' for number of days ', NumberOfDays(i) )
               ( (3) a, f(6) );
            END;
         END;
      ELSE  DO;
         PUT SKIP LIST( 'Error ' || FC.MsgNo
            || ' converting Date to Lilian Date' );
         END;
      /*********************************************************/
      /* Call CEEDYWK to return the day-of-the-week value      */
      /* (1 thru 7) for each calculated Lilian date.  Compare  */
      /* results to an array of expected returned values and   */
      /* issue an error message for any incorrect values.      */
      /*********************************************************/
      Call CEEDYWK ( Displaced_Lilian, WeekDay, FC );
      IF FBCHECK( FC, CEE000)  THEN  DO;
         IF  WeekDay ¬= ChkWeekDay(ii)  THEN  DO;
            PUT SKIP EDIT( 'Error in day of the week for ', ChrDate)
               ( a, a );
            END;
         END;
      ELSE  DO;
         PUT SKIP LIST( 'Error finding Day-of-Week');
         END;

      END;

   PUT SKIP LIST( '<<< Example CE77DAT complete');

END CE77DAT;
```

Figure 89. Calls to CEEDAYS, CEEDATE, and CEEDYWK in PL/I (Part 2 of 2)

Calls to CEECBLDY in COBOL

This example shows converting a 2-digit input date to a COBOL Integer date, adding 90 days to the Integer date, and converting the Integer format date back to a 4-digit year format using COBOL intrinsic functions.

```
CBL APOST
*Module/File Name: IGZTBDY
*************************************************
**                                             **
** Function: Invoke CEECBLDY callable service  **
** to convert date to COBOL Integer format.    **
** This service is used when using the         **
** Language Environment Century Window         **
** mixed with COBOL Intrinsic Functions.       **
**                                             **
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLDAYS.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  CHRDATE.
     05  CHRDATE-LENGTH      PIC  9(2) BINARY.
     05  CHRDATE-STRING      PIC X(50).
 01  PICSTR.
     05  PICSTR-LENGTH       PIC S9(4) BINARY.
     05  PICSTR-STRING       PIC X(50).
 01  INTEGER                 PIC S9(9) BINARY.
 01  NEWDATE                 PIC 9(8).
 01   FC                     PIC X(12).
 PROCEDURE DIVISION.
*************************************************
** Specify input date and length               **
*************************************************
     MOVE '1 January 00' to CHRDATE-STRING.
     MOVE 25 TO CHRDATE-LENGTH.
*************************************************
** Specify a picture string that describes     **
** input date, and the picture string's length.**
*************************************************
     MOVE 'ZD Mmmmmmmmmmmmmmz YY'
                          TO PICSTR-STRING.
     MOVE 23 TO PICSTR-LENGTH.
*************************************************
** Call CEECBLDY to convert input date to a    **
** COBOL Integer date                          **
*************************************************
     CALL 'CEECBLDY' USING CHRDATE, PICSTR,
                           INTEGER, FC.
*************************************************
** If CEECBLDY runs successfully, then compute **
** the date of the 90th day after the          **
** input date using Intrinsic Functions        **
*************************************************
     IF (FC = LOW-VALUE) THEN
         COMPUTE INTEGER = INTEGER + 90
         COMPUTE NEWDATE = FUNCTION
                        DATE-OF-INTEGER (INTEGER)
         DISPLAY NEWDATE ' is Integer day: ' INTEGER
     ELSE
         CONTINUE
     END-IF.

     GOBACK.
```

Figure 90. Calls to CEECBLDY in COBOL

Chapter 17. National Language Support Services

This chapter introduces the national language support services, which you use to set the national language, the country code, currency symbols, and decimal separators. It includes examples showing you how to query the default country code and change it, how to get the default date and time in the new country code, and how to convert the seconds to a timestamp. It also provides guidance for setting national language and country codes, including examples that show how national language services work in conjunction with date and time services.

Understanding the Basics

National language support services allow you to customize LE/VSE output (such as messages, RPTOPTS reports, RPTSTG reports, or dumps) for a given country by specifying the following:

- The language in which run-time messages, days of the week, and months are displayed and printed
- A country code that indicates the default date and time format, currency symbol, decimal separator, and thousands separator

Related Options and Services

Run-Time Options

COUNTRY	Sets default country
NATLANG	Sets initial national language

Callable Services

CEE5CTY	Sets default country
CEE5LNG	Sets national language
CEE5MCS	Gets default currency symbol
CEE5MDS	Gets default decimal separator
CEE5MTS	Gets default thousands separator
CEEFMDA	Gets default date format
CEEFMDT	Gets default date and time format
CEEFMTM	Gets default time format

Most of the tasks you perform with national language support services involve date and time services as well. See Chapter 16, “Using Date and Time Services,” on page 213 for a discussion of callable services for date and time calculations and see *LE/VSE Programming Reference* for syntax information for all callable services.

© Copyright IBM Corp. 1991, 2005

Setting the National Language

You can set the national language with the NATLANG (specify national language) run-time option or the CEE5LNG (set national language) callable service. The national language settings affect the error messages, month name, and day of the week name. Message translations are provided for the following languages:

UEN—Uppercase U.S. English
ENU—Mixed-case U.S. English
JPN—Japanese

Setting the Country Code

You can use the COUNTRY run-time option or the CEE5CTY callable service to set the current country code for your application. The country code determines the default formats used to display and print the date and timestamps in the reports generated by the:

- RPTSTG run-time option,
- RPTOPTS run-time option,
- CEE5DMP (dump) callable service.

Default values associated with the country code also describe the currency symbol, decimal separator, and thousands separator.

Because CEE5LNG and CEE5CTY allow you to maintain multiple national languages and country settings on separate LIFO stacks, you can easily reset the national language or alternate between different country settings. For example, if you want to ensure that a routine in your application outputs the date and time in a Japanese format, use CEE5CTY to query the current default setting and, if necessary, to set it to Japanese with CEE5CTY if some other country code is in effect. For sample user code, see *LE/VSE Programming Reference*.

The C language provides locales, which are UNIX structures that reflect different combinations of languages, cultural and territorial conventions, and codepages. Locale-sensitive C-language functions make use of values and formats in the current locale. There is a set of Language Environment locale callable services that exploit a subset of the C run-time interfaces for internationalized applications (see Chapter 18, “Locale Callable Services,” on page 257).

However, although C locale support and Language Environment callable services for national language support overlap functionally, they are completely independent of each other. Locale settings and the COUNTRY run-time option do not affect each other. Likewise, within Language Environment, locale callable services and the national language support callable services do not intersect. National language callable services derive values and formats only from defaults established by the COUNTRY run-time option or the CEE5CNTY service.

Note: LE/VSE does not currently support certain languages as national languages, so you would not be able to use CEE5LNG to set the national language to an unsupported language. You can, however, change the date and time format so that your English or Japanese banking application, for example, would display the default date and time format for an unsupported language. In general, you must use CEE5CTY to set the conventions for formatting date and time information.

Euro Support

The current country code determines the default currency symbol that will be returned by the CEE5MCS callable service. For countries in the European Union that have adopted the Euro as the legal tender, the currency symbol is represented as a hex string in the default country settings (for the country settings, refer to the manual see the Language Environment Programming Reference for country settings). The value is taken from a typical code page for the given country, but, of course, the actual graphical representation depends on the code page in use.

Language Environment supports the Euro as the default currency symbol in the following countries:
- Austria
- Belgium
- Finland
- France
- Germany
- Greece
- Ireland
- Italy
- Luxembourg
- The Netherlands
- Portugal
- Spain.

As more countries pass the Economic and monetary union convergence criteria and adopt the Euro as the legal currency, the default currency symbol will replace the national currency symbol with the Euro.

Combining National Language Support and Date and Time Services

To customize your applications for a particular country, use national language support services to query the current country code, which you then can use as input to the LE/VSE date and time callable services. For example, you could query the current country code with CEE5CTY and then use the returned value and CEEFMDT to get the default date and time format. When calling the CEEDATM (convert seconds to character timestamp) date and time service, you can use the string returned by CEEFMDT to specify the format of the convert seconds to character timestamp.

Calls to CEE5CTY, CEEFMDT, and CEEDATM in C

This example illustrates how you would query the default country code (CEE5CTY), change it to another country code (CEE5CTY), get the default date and time in the new country code (CEEFMDT), and convert the seconds to a timestamp (CEEDATM).

```
/*Module/File Name: EDCNLS */
/*******************************************************************/
/* FUNCTION                                                        */
/*            CEE5CTY   : query default country. set country to    */
/*                      : Germany.                                 */
/*            CEEFMDT   : get the German date and time format      */
/*            CEEDATM   : convert seconds to timestamp             */
/*                                                                 */
/* This example shows how to use several of the LE national        */
/* language support callable services. The current country is queried */
/* and changed to Germany. The default date and time for Germany is  */
/* obtained. CEEDATM is called to convert a large numeric value in   */
/* seconds to the timestamp 16.05.1988 19:01:01.                   */
/*******************************************************************/
#include <stdio.h>
#include <string.h>
#include <leawi.h>
#include <stdlib.h>
#include <ceeedcct.h>

int main(void) {

  _FEEDBACK fc;
  _INT4 function;
  _CHAR2 country, symbol;
  _CHAR80 date_pic;
  _FLOAT8 seconds;
  _VSTRING  picstr;
  _CHAR80 timestp;
  #define DE  "DE"
  #define BL  "  "

  printf ( "\n**********************************\n");
  printf ( "CESCNLS C Example is now in motion");
  printf ( "\n**********************************\n");
```

Figure 91. Querying and Setting the Country Code and Getting the Date and Time Format in C (Part 1 of 2)

```
  printf ( "\n**********************************\n");
  printf ( "CESCNLS C Example is now in motion");
  printf ( "\n**********************************\n");

  /****************************************************/
  /* Call CEE5CTY to query the current country setting */
  /****************************************************/
  function = 2;
  CEE5CTY(&function,country,&fc);
  if ( (_FBCHECK (fc , CEE000)) != 0 ) {
     printf("CEE5CTY failed with message number %d\n",fc.tok_msgno);
     exit(2999);
  }

  /****************************************************/
  /* Call CEE5CTY to set current country to Germany.   */
  /****************************************************/
  function = 3;
  CEE5CTY(&function,DE,&fc);
  if ( (_FBCHECK (fc , CEE000)) != 0 ) {
     printf("CEE5CTY failed with message number %d\n",fc.tok_msgno);
     exit(2999);
  }
  /************************************************************/
  /* Call CEEFMDT retrieve the default date and time format     */
  /************************************************************/
  CEEFMDT(BL,date_pic,&fc);
  if ( (_FBCHECK (fc , CEE000)) != 0 ) {
     printf("CEEFMDT failed with message number %d\n",fc.tok_msgno);
     exit(2999);
  }

  /************************************************************/
  /* Call CEEDATM to convert the number of seconds from 12:00AM */
  /* October 14, 1582 to 7:01PM May 16, 1988 to character       */
  /* format. The default date and time format matches that of   */
  /* the default country, Germany.                              */
  /************************************************************/
  seconds = 12799191661.986;
  strcpy(picstr.string,date_pic);
  picstr.length = strlen(picstr.string);
  CEEDATM ( &seconds , &picstr , timestp , &fc );
  if ( (_FBCHECK (fc , CEE000)) != 0 ) {
     printf("CEE5MDS failed with message number %d\n",fc.tok_msgno);
     exit(2999);
  }
  printf("Generated timestamp: %s",timestp);
  printf ("\n*********************\n");
  printf ("CESCNLS example ended.");
  printf ("\n*********************\n");
}
```

Figure 91. Querying and Setting the Country Code and Getting the Date and Time Format in C (Part 2 of 2)

Calls to CEE5CTY, CEEFMDT, and CEEDATM in COBOL

This example illustrates how you would query the default country code (CEE5CTY), change it to another country code (CEE5CTY), get the default date and time in the new country code (CEEFMDT), and convert the seconds to a timestamp (CEEDATM).

```
CBL LIB,APOST,RENT,OPTIMIZE
      *Module/file name: IGZTNLS
      ************************************************************
      **                                                        **
      **  CESCNLS - Call the following LE services:             **
      **                                                        **
      **              CEE5CTY  :  query default country         **
      **              CEEFMDT  :  obtain the default date and   **
      **                          time format                   **
      **              CEEDATM  :  convert seconds to timestamp **
      **                                                        **
      ** This example shows how to use several of the LE        **
      ** national language support callable services in a       **
      ** COBOL program.  The current country is queried, saved, **
      ** and then changed to Germany. The default date and time **
      ** for Germany is obtained. CEEDATM is called to          **
      ** convert a large numeric value in seconds to the        **
      ** timestamp 16.05.1988 19:01:01 (May 16, 1988 7:01PM.)   **
      **                                                        **
      ************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CESCNLS.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  SECONDS                 COMP-2.
       01  FUNCTN                  PIC S9(9) BINARY.
       01  COUNTRY                 PIC X(2).
       01  GERMANY                 PIC X(2) VALUE 'DE'.
       01  PICSTR.
           02  Vstring-length      PIC S9(4) BINARY.
           02  Vstring-text.
               03  Vstring-char        PIC X
                           OCCURS 0 TO 256 TIMES
                           DEPENDING ON Vstring-length
                              of PICSTR.
       01  TIMESTP                 PIC X(80).
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) BINARY.
       01  QUERY-COUNTRY-SETTING   PIC S9(9) BINARY VALUE 2.
       01  SET-COUNTRY-SETTING     PIC S9(9) BINARY VALUE 3.
       01  COUNTRY-PIC-STR         PIC X(80).
```

Figure 92. Querying and Setting the Country Code and Getting the Date and Time Format in COBOL (Part 1 of 2)

```
 PROCEDURE DIVISION.

 0001-BEGIN-PROCESSING.
     DISPLAY '*************************************'.
     DISPLAY 'CESCNLS COBOL example is now in motion. '.
     DISPLAY '*************************************'.
*************************************************************
*      Query Country Setting                                *
*************************************************************
     MOVE QUERY-COUNTRY-SETTING TO FUNCTN.
     CALL 'CEE5CTY' USING FUNCTN, COUNTRY, FC.
     IF NOT CEE000 of FC THEN
         DISPLAY 'Error ' Msg-No of FC
             ' in query of country setting'
     ELSE
*************************************************************
*        Call CEE5CTY to set country to Germany             *
*************************************************************
         MOVE SET-COUNTRY-SETTING TO FUNCTN
         MOVE GERMANY TO COUNTRY
         CALL 'CEE5CTY' USING FUNCTN, COUNTRY, FC
         IF NOT CEE000 of FC THEN
             DISPLAY 'Error ' Msg-No of FC
                 ' in setting country'
         ELSE
*************************************************************
*            Call CEEFMDT to get default date/time          *
*            format for Germany and verify format           *
*            against the published value.                   *
*************************************************************
             MOVE SPACE TO COUNTRY
             CALL 'CEEFMDT' USING COUNTRY, COUNTRY-PIC-STR, FC
             IF NOT CEE000 of FC THEN
                 DISPLAY 'Error getting default date/time'
                         ' format for Germany.'
             ELSE
*************************************************************
*                Call CEEDATM to convert the number of      *
*                seconds from October 14, 1582 12:00AM      *
*                to 16 May 1988 7:01PM to character format.*
*                The default date and time matches          *
*                that of the default country, Germany.      *
*************************************************************
                 MOVE 12799191661.986 TO SECONDS
                 COMPUTE Vstring-length OF PICSTR =
                   FUNCTION MIN( LENGTH OF COUNTRY-PIC-STR, 256 )
                 MOVE COUNTRY-PIC-STR TO Vstring-text of PICSTR
                 CALL 'CEEDATM' USING SECONDS, PICSTR,
                                      TIMESTP, FC
                 IF CEE000 of FC THEN
                     DISPLAY 'Generated timestamp is: ' TIMESTP
                 ELSE
                     DISPLAY 'Error ' Msg-No of FC
                         ' generating timestamp'
                 END-IF
             END-IF
             DISPLAY '**********************'
             DISPLAY 'COBOL NLS example ended'
             DISPLAY '**********************'
         END-IF
     END-IF.

     GOBACK.
```

Figure 92. Querying and Setting the Country Code and Getting the Date and Time Format in COBOL (Part 2 of 2)

Example Using CEE5CTY, CEEFMDT, and CEEDATM in PL/I

```
*PROCESS MACRO;
      /*Module/File Name: IBMNLS
      /*********************************************************/
      /*                                                       */
      /* Function      CEE5CTY   : query default country       */
      /*               CEEFMDT   : obtain the default date and  */
      /*                           time format                 */
      /*               CEEDATM   : convert seconds to timestamp */
      /*                                                       */
      /* This example shows how to use several of the LE       */
      /* national language support callable services in a      */
      /* PL/I program. The current country is queried, saved,  */
      /* and then changed to Germany. The default date and     */
      /* time for Germany is obtained. CEEDATM is called to    */
      /* convert a large numeric value in seconds to the       */
      /* timestamp 16.05.1988 19:01:01 (May 16, 1988 7:01PM).  */
      /*                                                       */
      /*********************************************************/
    CESCNLS: PROC OPTIONS(MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL FUNCTN          INT4;
    DCL QUERY_COUNTRY   INT4 INIT(2);
    DCL SET_COUNTRY     INT4 INIT(3);
    DCL SECONDS         FLOAT8;
    DCL COUNTRY         CHARACTER ( 2 );
    DCL GERMANY         CHARACTER ( 2 )INIT ('DE');
    DCL 01 FC           FEEDBACK;
    DCL TIMESTP         CHAR(80);
    DCL PICSTR          CHAR(80);
    DCL PIC_VSTR        CHAR(255) VARYING;

    /*********************************************************/
    /* Query country setting                                 */
    /*********************************************************/
    FUNCTN = QUERY_COUNTRY;
    CALL CEE5CTY ( FUNCTN, COUNTRY, FC );
    IF  FBCHECK( FC, CEE000)  THEN  DO;
       /***************************************************/
       /* Call CEE5CTY to set country to Germany          */
       /***************************************************/
       FUNCTN = SET_COUNTRY;
       COUNTRY = GERMANY;
       CALL CEE5CTY ( FUNCTN, COUNTRY, FC );
       IF &#170; FBCHECK( FC, CEE000)  THEN  DO;
          PUT SKIP LIST('Error ' || FC.MsgNo || ' in setting country');
          END;
       ELSE DO;
          /*****************************************************/
          /* Call CEEFMDT to get default date/time format for   */
          /* Germany and verify format against published value. */
          /*****************************************************/
          COUNTRY = ' ';
          CALL CEEFMDT ( COUNTRY, PICSTR, FC );
          IF &#170; FBCHECK( FC, CEE000)  THEN  DO;
             PUT SKIP LIST( 'Error ' || FC.MsgNo
                || ' getting default date/time format for Germany.');
             END;
```

Figure 93. Querying and Setting the Country Code and Getting the Date and Time Format in PL/I (Part 1 of 2)

```
    ELSE DO;
           /**************************************************/
           /* Call CEEDATM to convert the number representing */
           /* the number of seconds from October 14, 1582     */
           /* 12:00AM to 16 May 1988 7:01PM to character      */
           /* format. The default date and time format        */
           /* matches that of the default country, Germany.   */
           /**************************************************/
           SECONDS = 12799191661.986;
           PIC_VSTR = PICSTR;
           CALL CEEDATM ( SECONDS, PIC_VSTR, TIMESTP, FC );
           IF FBCHECK( FC, CEE000)  THEN  DO;
              PUT SKIP EDIT ('Generated timestamp is ',
                             TIMESTP) (A, A);
              END;
           ELSE  DO;
              PUT SKIP LIST ('Error ' || FC.MsgNo
                 || ' generating timestamp');
              END;
           END;
        END;
     END;
   ELSE  DO;
     PUT SKIP LIST( 'Error ' || FC.MsgNo || ' querying country code');
     END;

END CESCNLS;
```

Figure 93. Querying and Setting the Country Code and Getting the Date and Time Format in PL/I (Part 2 of 2)

Chapter 18. Locale Callable Services

This chapter describes how to use the LE/VSE locale callable services to internationalize your applications, and includes examples that show how locale callable services work in conjunction with each other. Locale callable services do not affect, nor are they affected by, LE/VSE callable services or the COUNTRY or NATLANG run-time options.

LE/VSE locale support adheres to the standards used by C. For detailed information on these standards, locales, and charmaps, see *LE/VSE C Run-Time Programming Guide*.

Note: LE/VSE locale callable services use the LE/VSE C run-time library. As the C run-time library must be addressed in 31-bit addressing mode, phases containing calls to locale callable services must be link-edited with AMODE=31.

Understanding the Basics

Locale callable services allow you to customize culturally-sensitive output for a given national language, country, and codeset by specifying a locale name.

Related Services

Callable Services

CEEFMON Formats monetary string
CEEFTDS Formats date and time into a character string
CEELCNV Query locale numeric conventions
CEEQDTC Queries locale, date, and time conventions
CEEQRYL Queries the active locale environment
CEESCOL Compares the collation weights of two strings
CEESETL Sets the locale operating environment
CEESTXF Transforms string characters into collation weights

See *LE/VSE Programming Reference* for syntax information.

Although C routines can use the locale callable services, it is recommended that they use the equivalent native C library services instead for portability across platforms. Table 42 shows the LE/VSE locale callable services and the equivalent C library routines.

Table 42. LE/VSE Locale Callable Services and Equivalent C Library Routines

LE/VSE Locale Callable Service	C Library Routine
CEEFMON	`strfmon()`
CEEFTDS	`strftime()`
CEELCNV	`localeconv()`
CEEQDTC	`localdtconv()`
CEEQRYL	`setlocale()`
CEESCOL	`strcoll()`

© Copyright IBM Corp. 1991, 2005

Table 42. LE/VSE Locale Callable Services and Equivalent C Library Routines (continued)

LE/VSE Locale Callable Service	C Library Routine
CEESETL	`setlocale()`
CEESTXF	`strxfrm()`

Developing Internationalized Applications

Locale callable services define environment control variables that you can set to establish language-specific information and preferences for an application. Locale callable services also provide a means for establishing global preferences, such as `setlocale()`, locale management services, and locale-dependent interfaces to the application.

Locale callable services allow you to develop applications that can be used in multiple countries, because they can function with specific language and cultural conventions. Such applications are referred to as internationalized applications. These applications have no built-in assumptions with respect to the language, culture, or conventions of their users or the data they process. Instead, language and cultural information is set at run time, a process called localization. Thus, the application processes data provided specifically for a certain locale. In LE/VSE, localization occurs at the enclave level.

Examples of Using Locale Callable Services

The examples in this section illustrate how you can use locale callable services.

Examples Illustrating Calls to CEEFMON

The following examples illustrate calls to CEEFMON to convert a numeric value to a monetary string using a specified format.

Calls to CEEFMON in COBOL

```
 CBL LIB,APOST,RMODE(ANY)
*Module/File Name: IGZTFMON
*************************************************
*  Example for callable service CEEFMON        *
*   Function: Convert a numeric value to a      *
*             monetary string using specified  *
*             format passed as parameter.      *
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  COBFMON.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  Monetary       COMP-2.
* OMIT is a dummy parameter used across LE call.
 01  OMIT           COMP-2.
 01  Max-Size       PIC S9(9)  BINARY.
 01  Format-Mon.
     02  FM-Length  PIC S9(4)  BINARY.
     02  FM-String  PIC X(256).
 01  Output-Mon.
     02  OM-Length  PIC S9(4)  BINARY.
     02  OM-String  PIC X(60).
 01  Length-Mon     PIC S9(9)  BINARY.
 01  FC.
     02  Condition-Token-Value.
     COPY CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) BINARY.
             04  Msg-No      PIC S9(4) BINARY.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) BINARY.
             04  Cause-Code  PIC S9(4) BINARY.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info           PIC S9(9) BINARY.
*
 PROCEDURE DIVISION.
*  Set up numeric value
     MOVE 12345.62 TO Monetary.
     MOVE 60 TO Max-Size.
     MOVE 2 TO FM-Length.
     MOVE '%i' TO FM-String (1:FM-Length).

*  Call CEEFMON to convert numeric value
     CALL 'CEEFMON' USING OMIT, Monetary,
                          Max-Size, Format-Mon
                          Output-Mon, Length-Mon,
                          FC.

*  Check feedback code and display result
     IF Severity > 0
        DISPLAY 'Call to CEEFMON failed. ' Msg-No
     ELSE
        DISPLAY 'International format is '
                OM-String(1:OM-Length)
     END-IF.

     STOP RUN.
 END PROGRAM COBFMON.
```

Figure 94. Calls to CEEFMON in COBOL

Calls to CEEFMON in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMFMON                               */
 /****************************************************/
 /* Example for callable service CEEFMON              */
 /* Function: Convert a numeric value to a monetary  */
 /*  string using specified format passed as parm    */
 /****************************************************/

 PLIFMON: PROC OPTIONS(MAIN);

 %INCLUDE CEEIBMAW; /* ENTRY defs, macro defs        */
 %INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants   */
 %INCLUDE CEEIBMLC; /* Locale category constants     */

 /* CEEFMON service call arguments */
 DCL MONETARY     FLOAT8;           /* input value   */
 DCL MAXSIZE_FMON INT4;             /* output size   */
 DCL FORMAT_FMON  CHAR(256) VARYING;/* format spec   */
 DCL RESULT_FMON  INT4;             /* result status */
 DCL OUTPUT_FMON  CHAR(60) VARYING; /* output string */

 DCL 01 FC        FEEDBACK;

   MONETARY = 12345.62; /* monetary numeric value    */
   MAXSIZE_FMON = 60;   /* max char length returned  */
   FORMAT_FMON = '%i';  /* international currency    */

   CALL CEEFMON ( *, /* optional argument           */
     MONETARY ,      /* input, 8 byte floating point */
     MAXSIZE_FMON,   /* maximum size of output string*/
     FORMAT_FMON,    /* conversion request           */
     OUTPUT_FMON,    /* string returned by CEEFMON   */
     RESULT_FMON,    /* no. of chars in OUTPUT_FMON  */
     FC );           /* feedback code structure      */

   IF FC.Severity > 0 THEN
     DO;
       /* FBCHECK macro used (defined in CEEIBMCT) */
       IF FBCHECK( FC, CEE3VM ) THEN
         DISPLAY ( 'Invalid input '||MONETARY );
       ELSE
         DISPLAY ('CEEFMON not completed '||FC.MsgNo );
       STOP;
     END;
   ELSE
     DO;
       PUT SKIP LIST(
         'International Format '||OUTPUT_FMON );
     END;

 END PLIFMON;
```

Figure 95. Calls to CEEFMON in PL/I

Examples Illustrating Calls to CEEFTDS

The following examples illustrate calls to CEEFTDS to convert a numeric time and date to a string using a specified format.

Calls to CEEFTDS in COBOL

```
 CBL LIB,APOST,RMODE(ANY)
*Module/File Name: IGZTFTDS
*************************************************
*  Example for callable service CEEFTDS        *
*  Function: Convert numeric time and date     *
*            values to a string using specified *
*            format string and locale format    *
*            conversions.                       *
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  MAINFTDS.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* Use TD-Struct for CEEFTDS calls
 COPY CEEIGZTD.
*

 PROCEDURE DIVISION.
*  Subroutine needed for pointer addressing
     CALL 'COBFTDS' USING TD-Struct.

     STOP RUN.
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  COBFTDS.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* Use Locale category constants
 COPY CEEIGZLC.
*
* OMIT is a dummy parameter used across LE call.
 01  OMIT           COMP-2.
 01  Ptr-FTDS  POINTER.
 01  Output-FTDS.
     02  O-Length  PIC S9(4)  BINARY.
     02  O-String  PIC X(72).
 01  Format-FTDS.
     02  F-Length  PIC S9(4)  BINARY.
     02  F-String  PIC X(64).
 01  Max-Size  PIC S9(9)  BINARY.
 01  FC.
     02  Condition-Token-Value.
     COPY CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity  PIC S9(4) BINARY.
             04  Msg-No    PIC S9(4) BINARY.
         03  Case-2-Condition-ID
                REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) BINARY.
             04  Cause-Code  PIC S9(4) BINARY.
         03  Case-Sev-Ctl  PIC X.
         03  Facility-ID   PIC XXX.
     02  I-S-Info  PIC S9(9) BINARY.
 LINKAGE SECTION.
* Use TD-Struct for calls to CEEFTDS
 COPY CEEIGZTD.
*
```

Figure 96. Calls to CEEFTDS in COBOL (Part 1 of 2)

```
 PROCEDURE DIVISION USING TD-Struct.
*  Set up time and date values
     MOVE 1 TO TM-Sec.
     MOVE 2 TO TM-Min.
     MOVE 3 TO TM-Hour.
     MOVE 9 TO TM-Day.
     MOVE 11 TO TM-Mon.
     MOVE 94 TO TM-Year.
     MOVE 5 TO TM-Wday.
     MOVE 344 TO TM-Yday.
     MOVE 1 TO TM-Is-DLST.

*  Set up format string for CEEFTDS call
     MOVE 72 TO Max-Size.
     MOVE 36 TO F-Length.
     MOVE 'Today is %A, %b %d  Time:  %I:%M  %p'
        TO F-String (1:F-Length).

*  Set up pointer to structure for CEEFTDS call
     SET Ptr-FTDS TO ADDRESS OF TD-Struct.

*  Call CEEFTDS to convert numeric values
     CALL 'CEEFTDS' USING OMIT, Ptr-FTDS,
                          Max-Size, Format-FTDS,
                          Output-FTDS, FC.

*  Check feedback code and display result
     IF Severity = 0
        DISPLAY 'Format ' F-String (1:F-Length)
        DISPLAY 'Result ' O-String (1:O-Length)
     ELSE
        DISPLAY 'Call to CEEFTDS failed. ' Msg-No
     END-IF.

     EXIT PROGRAM.
 END PROGRAM COBFTDS.
*
 END PROGRAM MAINFTDS.
```

Figure 96. Calls to CEEFTDS in COBOL (Part 2 of 2)

Calls to CEEFTDS in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMFTDS                          */
 /****************************************************/
 /* Example for callable service CEEFTDS             */
 /* Function: Convert numeric time and date values   */
 /*  to a string based on a format specification     */
 /*  string parameter and locale format conversions  */
 /****************************************************/

 PLIFTDS: PROC OPTIONS(MAIN);

 %INCLUDE CEEIBMAW; /* ENTRY defs, macro defs        */
 %INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants   */
 %INCLUDE CEEIBMLC; /* Locale category constants     */
 %INCLUDE CEEIBMTD; /* TD_STRUCT for CEEFTDS calls   */

 /* use explicit pointer to local TD_STRUCT structure*/
 DCL TIME_AND_DATE POINTER INIT(ADDR(TD_STRUCT));

 /* CEEFTDS service call arguments */
 DCL MAXSIZE_FTDS BIN FIXED(31); /* OUTPUT_FTDS size */
 DCL FORMAT_FTDS CHAR(64) VARYING;  /* format string */
 DCL OUTPUT_FTDS CHAR(72) VARYING;  /* output string */

 DCL 01 FC       FEEDBACK;

   /* specify numeric input fields for conversion    */
   TD_STRUCT.TM_SEC=1;   /* seconds after min (0-61) */
   TD_STRUCT.TM_MIN=2;   /* minutes after hour (0-59)*/
   TD_STRUCT.TM_HOUR=3; /* hours since midnight(0-23)*/
   TD_STRUCT.TM_MDAY=9;  /* day of the month (1-31)  */
   TD_STRUCT.TM_MON=11;  /* months since Jan(0-11)   */
   TD_STRUCT.TM_YEAR=94; /* years since 1900         */
   TD_STRUCT.TM_WDAY=5;  /* days since Sunday (0-6)  */
   TD_STRUCT.TM_YDAY=344;/* days since Jan 1 (0-365) */
   TD_STRUCT.TM_ISDST=1; /* Daylight Saving Time flag*/

   /* specify format string for CEEFTDS call         */
   FORMAT_FTDS = 'Today is %A, %b %d  Time: %I:%M %p';

   MAXSIZE_FTDS = 72; /* specify output string size  */

   CALL CEEFTDS ( *, TIME_AND_DATE, MAXSIZE_FTDS,
                  FORMAT_FTDS, OUTPUT_FTDS, FC );

   /* FBCHECK macro used (defined in CEEIBMCT) */
   IF FBCHECK( FC, CEE000 ) THEN
     DO; /* CEEFTDS call is successful */
       PUT SKIP LIST('Format '||FORMAT_FTDS );
       PUT SKIP LIST('Results in '||OUTPUT_FTDS );
     END;
   ELSE
     DISPLAY ( 'Format '||FORMAT_FTDS||
               ' Results in '||FC.MsgNo );

 END PLIFTDS;
```

Figure 97. Calls to CEEFTDS in PL/I

Examples Illustrating Calls to CEELCNV and CEESETL

The following examples illustrate calls to CEELCNV to retrieve the numeric and monetary format for the default locale, and to CEESETL to set the locale.

Calls to CEELCNV and CEESETL in COBOL

```
 CBL LIB,APOST,RMODE(ANY)
*Module/File Name: IGZTLCNV
*************************************************
** Example for callable service CEELCNV       **
**  Function: Retrieve numeric and monetary    **
**             format for default locale and   **
**             print an item.                  **
**            Set locale to France, retrieve   **
**             structure, and print an item.   **
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  MAINLCNV.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*************************************************
** Use Locale NM-Struct for CEELCNV calls      **
*************************************************
 COPY CEEIGZNM.
*
 PROCEDURE DIVISION.
*************************************************
**  Subroutine needed for addressing           **
*************************************************
     CALL 'COBLCNV' USING NM-Struct.

     STOP RUN.
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  COBLCNV.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* OMIT is a dummy parameter used across LE call.
 01  OMIT           COMP-2.
 01  Locale-Name.
     02  LN-Length  PIC S9(4) BINARY.
     02  LN-String  PIC X(256).
*************************************************
** Use Locale category constants              **
*************************************************
 COPY CEEIGZLC.
*
 01  FC.
     02  Condition-Token-Value.
     COPY CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) BINARY.
             04  Msg-No      PIC S9(4) BINARY.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) BINARY.
             04  Cause-Code  PIC S9(4) BINARY.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) BINARY.
 LINKAGE SECTION.
```

Figure 98. Calls to CEELCNV and CEESETL in COBOL (Part 1 of 2)

```
*************************************************
    ** Use Locale NM-Struct for CEELCNV calls      **
    *************************************************
     COPY CEEIGZNM.
    *
     PROCEDURE DIVISION USING NM-Struct.
    *************************************************
    **  Call CEELCNV to retrieve values for locale**
    *************************************************
         CALL 'CEELCNV' USING OMIT,
                        ADDRESS OF NM-Struct, FC.

    *************************************************
    **  Check feedback code and display result     **
    *************************************************
         IF Severity = 0 THEN
            DISPLAY 'Default decimal point is '
            DECIMAL-PT-String(1:DECIMAL-PT-Length)
         ELSE
            DISPLAY 'Call to CEELCNV failed. ' Msg-No
         END-IF.

    *************************************************
    ** Set up locale for France                    **
    *************************************************
         MOVE 4 TO LN-Length.
         MOVE 'FRAN' TO LN-String (1:LN-Length).

    *************************************************
    **  Call CEESETL to set monetary locale        **
    *************************************************
         CALL 'CEESETL' USING Locale-Name,
               LC-MONETARY, FC.

    *************************************************
    **  Call CEESETL to set numeric locale         **
    *************************************************
         CALL 'CEESETL' USING Locale-Name,
               LC-NUMERIC, FC.

    *************************************************
    **  Check feedback code and call CEELCNV again **
    *************************************************
         IF Severity = 0
            CALL 'CEELCNV' USING OMIT,
                           ADDRESS OF NM-Struct, FC
            IF Severity > 0
               DISPLAY 'Call to CEELCNV failed. '
                       Msg-No
            ELSE
               DISPLAY 'French decimal point is '
               DECIMAL-PT-String(1:DECIMAL-PT-Length)
            END-IF
         ELSE
            DISPLAY 'Call to CEESETL failed. ' Msg-No
         END-IF.

         EXIT PROGRAM.
     END PROGRAM COBLCNV.
    *
     END PROGRAM MAINLCNV.
```

Figure 98. Calls to CEELCNV and CEESETL in COBOL (Part 2 of 2)

Calls to CEELCNV and CEESETL in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMLCNV                          */
 /****************************************************/
 /* Example for callable service CEELCNV             */
 /* Function: Retrieve numeric and monetary format   */
 /*  structure for default locale and print an item. */
 /*  Set locale to France, retrieve structure and    */
 /*  print an item.                                  */
 /****************************************************/

 PLILCNV: PROC OPTIONS(MAIN);

 %INCLUDE CEEIBMAW; /* ENTRY defs, macro defs        */
 %INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants   */
 %INCLUDE CEEIBMLC; /* Locale category constants     */
 %INCLUDE CEEIBMNM; /* NM_STRUCT for CEELCNV calls   */

 /* use explicit pointer for local NM_STRUCT struct  */
 DCL NUM_AND_MON POINTER INIT(ADDR(NM_STRUCT));

 /* CEESETL service call arguments */
 DCL LOCALE_NAME CHAR(256) VARYING;

 DCL 01 FC       FEEDBACK;

   /* retrieve structure for default locale */
   CALL CEELCNV ( *, NUM_AND_MON, FC );

   PUT SKIP LIST('Default DECIMAL_POINT is ',
                NM_STRUCT.DECIMAL_POINT);

   /* set locale for France */
   LOCALE_NAME = 'FRAN';

   /* use LC_NUMERIC category const from CEEIBMLC */
   CALL CEESETL ( LOCALE_NAME, LC_NUMERIC,  FC );

   /* use LC_MONETARY category const from CEEIBMLC */
   CALL CEESETL ( LOCALE_NAME, LC_MONETARY,  FC );

   /* FBCHECK macro used (defined in CEEIBMCT) */
   IF FBCHECK( FC, CEE000 ) THEN
     DO;
       /* retrieve active NM_STRUCT, France Locale */
       CALL CEELCNV ( *, NUM_AND_MON, FC );

       PUT SKIP LIST('French DECIMAL_POINT is ',
                    NM_STRUCT.DECIMAL_POINT);
     END;

 END PLILCNV;
```

Figure 99. Calls to CEELCNV and CEESETL in PL/I

Examples Illustrating Calls to CEEQDTC and CEESETL

The following examples illustrate calls to CEEQDTC to retrieve the date and time conventions, and to CEESETL to set the locale.

Calls to CEEQDTC and CEESETL in COBOL

```
 CBL LIB,APOST,RMODE(ANY)
*Module/File Name: IGZTQDTC
*************************************************
*  Example for callable service CEEQDTC         *
*  MAINQDTC - Retrieve date and time convention  *
*             structures for two countries and   *
*             compare an item.                   *
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  MAINQDTC.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* Use DTCONV structure for CEEQDTC calls
 COPY CEEIGZDT.
*
 PROCEDURE DIVISION.
*  Subroutine needed for addressing
     CALL 'COBQDTC' USING DTCONV.

     STOP RUN.
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  COBQDTC.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
* OMIT is a dummy parameter used across LE call.
 01  OMIT           COMP-2.
 01  Locale-Name.
     02  LN-Length  PIC S9(4) BINARY.
     02  LN-String  PIC X(256).
* Use Locale category constants
 COPY CEEIGZLC.
*
 01  Test-Length1 PIC S9(4) BINARY.
 01  Test-String1 PIC X(80).
 01  Test-Length2 PIC S9(4) BINARY.
 01  Test-String2 PIC X(80).
 01  FC.
     02  Condition-Token-Value.
     COPY CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity  PIC S9(4) BINARY.
             04  Msg-No    PIC S9(4) BINARY.
         03  Case-2-Condition-ID
               REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) BINARY.
             04  Cause-Code  PIC S9(4) BINARY.
         03  Case-Sev-Ctl  PIC X.
         03  Facility-ID   PIC XXX.
     02  I-S-Info  PIC S9(9) BINARY.
*
 LINKAGE SECTION.
* Use Locale structure DTCONV for CEEQDTC calls
 COPY CEEIGZDT.
*
```

Figure 100. Calls to CEEQDTC and CEESETL in COBOL (Part 1 of 2)

```
 PROCEDURE DIVISION USING DTCONV.
*  Set up locale for France
     MOVE 4 TO LN-Length.
     MOVE 'FFEY' TO LN-String (1:LN-Length).

*  Call CEESETL to set all locale categories
     CALL 'CEESETL' USING Locale-Name, LC-ALL,
                          FC.

*  Check feedback code
     IF Severity > 0
        DISPLAY 'Call to CEESETL failed. ' Msg-No
        EXIT PROGRAM
     END-IF.

*  Call CEEQDTC for French values
     CALL 'CEEQDTC' USING OMIT,
                          ADDRESS OF DTCONV, FC.

*  Check feedback code
     IF Severity > 0
        DISPLAY 'Call to CEEQDTC failed. ' Msg-No
        EXIT PROGRAM
     END-IF.

*  Save date and time format for FFEY locale
     MOVE D-T-FMT-Length IN DTCONV TO Test-Length1
     MOVE D-T-FMT-String IN DTCONV TO Test-String1
*  Set up locale for French Canadian
     MOVE 4 TO LN-Length.
     MOVE 'FCEY' TO LN-String (1:LN-Length).
*  Call CEESETL to set locale for all categories
     CALL 'CEESETL' USING Locale-Name, LC-ALL,
                          FC.

*  Check feedback code
     IF Severity > 0
        DISPLAY 'Call to CEESETL failed. ' Msg-No
        EXIT PROGRAM
     END-IF.

*  Call CEEQDTC again for French Canadian values
     CALL 'CEEQDTC' USING OMIT,
                          ADDRESS OF DTCONV, FC.

*  Check feedback code and display results
     IF Severity = 0
*  Save date and time format for FCEY locale
        MOVE D-T-FMT-Length IN DTCONV
                            TO Test-Length2
        MOVE D-T-FMT-String IN DTCONV
                            TO Test-String2

        IF Test-String1(1:Test-Length1) =
           Test-String2(1:Test-Length2)
           DISPLAY 'Same date and time format.'
        ELSE
           DISPLAY 'Different formats.'
           DISPLAY  Test-String1(1:Test-Length1)
           DISPLAY  Test-String2(1:Test-Length2)
        END-IF
     ELSE
        DISPLAY 'Call to CEEQDTC failed. ' Msg-No
     END-IF.

     EXIT PROGRAM.
 END PROGRAM COBQDTC.
*
 END PROGRAM MAINQDTC.
```

Figure 100. Calls to CEEQDTC and CEESETL in COBOL (Part 2 of 2)

Calls to CEEQTDC and CEESETL in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMQDTC                         */
 /****************************************************/
 /* Example for callable service CEEQDTC            */
 /* Function: Retrieve date and time convention     */
 /*  structures for two countries, compare an item.  */
 /****************************************************/

 PLIQDTC: PROC OPTIONS(MAIN);

 %INCLUDE CEEIBMAW; /* ENTRY defs, macro defs       */
 %INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants   */
 %INCLUDE CEEIBMLC; /* Locale category constants     */
 %INCLUDE CEEIBMDT; /* DTCONV for CEEQDTC calls      */

 /* use explicit pointer to local DTCONV structure */
 DCL LOCALDT POINTER INIT(ADDR(DTCONV));

 /* CEESETL service call arguments */
 DCL LOCALE_NAME CHAR(256) VARYING;

 DCL 1 DTCONVC LIKE DTCONV; /* Def Second Structure  */

 DCL 1 FC    FEEDBACK;

   /* set locale with IBM default for France         */
   LOCALE_NAME = 'FFEY';   /* or Fr_FR.IBM-1047      */

   /* use LC_ALL category constant from CEEIBMLC */
   CALL CEESETL ( LOCALE_NAME, LC_ALL,  FC );

   /* retrieve date and time structure, France Locale*/
   CALL CEEQDTC ( *, LOCALDT, FC );

   /* set locale with French Canadian(FCEY) defaults */
   /* literal constant -1 used to set all categories */
   CALL CEESETL ( 'FCEY', -1,  FC );

   /* retrieve date and time tables for French Canada*/
   /* example of temp pointer used for service call  */
   CALL CEEQDTC ( *, ADDR(DTCONVC), FC );

   /* compare date and time formats for two countries*/
   IF DTCONVC.D_T_FMT = DTCONV.D_T_FMT THEN
     DO;
       PUT SKIP LIST('Countries have same D_T_FMT' );
     END;
   ELSE
     DO;
       PUT SKIP LIST('Date and Time Format ',
                     DTCONVC.D_T_FMT||' vs '||
                     DTCONV.D_T_FMT );
     END;

 END PLIQDTC;
```

Figure 101. Calls to CEEQTDC and CEESETL in PL/I

Examples Illustrating Calls to CEESCOL

The following examples illustrate calls to CEESCOL to compare the collation of two character strings.

Calls to CEESCOL in COBOL

```
 CBL LIB,APOST,RMODE(ANY)
*Module/File Name: IGZTSCOL
*************************************************
*  Example for callable service CEESCOL         *
*   COBSCOL - Compare two character strings     *
*              and print the result.            *
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  COBSCOL.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
* OMIT is a dummy parameter used across LE call.
 01  OMIT           COMP-2.
 01  String1.
     02  Str1-Length  PIC S9(4) BINARY.
     02  Str1-String.
         03  Str1-Char  PIC X
                        OCCURS 0 TO 256 TIMES
                        DEPENDING ON Str1-Length.
 01  String2.
     02  Str2-Length  PIC S9(4) BINARY.
     02  Str2-String.
         03  Str2-Char  PIC X
                        OCCURS 0 TO 256 TIMES
                        DEPENDING ON Str2-Length.
 01  Result  PIC S9(9)  BINARY.
 01  FC.
     02  Condition-Token-Value.
     COPY CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) BINARY.
             04  Msg-No      PIC S9(4) BINARY.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) BINARY.
             04  Cause-Code  PIC S9(4) BINARY.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) BINARY.
*
 PROCEDURE DIVISION.
*************************************************
*  Set up two strings for comparison
*************************************************
     MOVE 9 TO Str1-Length.
     MOVE '12345a789'
        TO Str1-String (1:Str1-Length)
     MOVE 9 TO Str2-Length.
     MOVE '12346$789'
        TO Str2-String (1:Str2-Length)

*************************************************
*  Call CEESCOL to compare the strings
*************************************************
     CALL 'CEESCOL' USING OMIT, String1,
                          String2, Result, FC.
```

Figure 102. Calls to CEESCOL in COBOL (Part 1 of 2)

```
***************************************************
 *  Check feedback code
 ***************************************************
      IF Severity > 0
         DISPLAY 'Call to CEESCOL failed. ' Msg-No
         STOP RUN
      END-IF.

 ***************************************************
 *  Check result of compare
 ***************************************************
      EVALUATE TRUE
         WHEN Result < 0
            DISPLAY '1st string < 2nd string.'
         WHEN Result > 0
            DISPLAY '1st string > 2nd string.'
         WHEN OTHER
            DISPLAY 'Strings are identical.'
      END-EVALUATE.

      STOP RUN.
  END PROGRAM COBSCOL.
```

Figure 102. Calls to CEESCOL in COBOL (Part 2 of 2)

Calls to CEESCOL in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMSCOL                            */
 /****************************************************/
 /* Example for callable service CEESCOL               */
 /* Function: Compare two character strings and        */
 /*  print the result.                                 */
 /****************************************************/

 PLISCOL: PROC OPTIONS(MAIN);

 %INCLUDE CEEIBMAW; /* ENTRY defs, macro defs for LE */
 %INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants   */
 %INCLUDE CEEIBMLC; /* Locale category constants     */

 /* CEESCOL service call arguments */
 DCL STRING1 CHAR(256) VARYING;/* first string       */
 DCL STRING2 CHAR(256) VARYING;/* second string      */
 DCL RESULT_SCOL BIN FIXED(31);/* result of compare  */

 DCL 01 FC   FEEDBACK;

   STRING1 = '12345a789';
   STRING2 = '12346$789';

   CALL CEESCOL( *, STRING1, STRING2, RESULT_SCOL,FC);

   /* FBCHECK macor used (defined in CEEIBMCT) */
   IF FBCHECK( FC, CEE3T1 ) THEN
     DO;
       DISPLAY ('CEESCOL not completed '||FC.MsgNo );
       STOP;
     END;

   SELECT;
   WHEN( RESULT_SCOL < 0 )
     PUT SKIP LIST(
       '"firststring" is less than "secondstring" ');
   WHEN( RESULT_SCOL > 0 )
     PUT SKIP LIST(
     '"firststring" is greater than "secondstring" ');
   OTHERWISE
     PUT SKIP LIST( 'Strings are identical' );
   END;  /* END SELECT */

 END PLISCOL;
```

Figure 103. Calls to CEESCOL in PL/I

Examples Illustrating Calls to CEESETL and CEEQRYL

The following examples illustrate calls to CEESETL to set the locale, and to CEEQRYL to retrieve locale time information.

Calls to CEESETL and CEEQRYL in COBOL

```
 CBL LIB,APOST,RMODE(ANY)
*Module/File Name: IGZTSETL
*************************************************
*  Example for callable service CEESETL        *
*   COBSETL - Set all global locale environment *
*              categories to country Sweden.    *
*             Query one category.               *
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  COBSETL.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  Locale-Name.
     02  LN-Length  PIC S9(4) BINARY.
     02  LN-String  PIC X(256).
 01  Locale-Time.
     02  LT-Length  PIC S9(4) BINARY.
     02  LT-String  PIC X(256).
* Use Locale category constants
 COPY CEEIGZLC.
*
 01  FC.
     02  Condition-Token-Value.
     COPY CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) BINARY.
             04  Msg-No      PIC S9(4) BINARY.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) BINARY.
             04  Cause-Code  PIC S9(4) BINARY.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) BINARY.
*
 PROCEDURE DIVISION.
*************************************************
*  Set up locale name for Sweden
*************************************************
     MOVE 14 TO LN-Length.
     MOVE 'Sv_SE.IBM-1047'
             TO LN-String (1:LN-Length).

*************************************************
*  Set all locale categories to Sweden
*  Use LC-ALL category constant from CEEIGZLC
*************************************************
     CALL 'CEESETL' USING Locale-Name, LC-ALL,
                          FC.

*************************************************
*  Check feedback code
*************************************************
     IF Severity > 0
        DISPLAY 'Call to CEESETL failed. ' Msg-No
        STOP RUN
     END-IF.
```

Figure 104. Calls to CEESETL and CEEQRYL in COBOL (Part 1 of 2)

```
 ************************************************
 *  Retrieve active locale for LC-TIME category
 ************************************************
      CALL 'CEEQRYL' USING LC-TIME, Locale-Time,
                           FC.

 ************************************************
 *  Check feedback code and correct locale
 ************************************************
      IF Severity = 0
         IF LT-String(1:LT-Length) =
                 LN-String(1:LN-Length)
            DISPLAY 'Successful query.'
         ELSE
            DISPLAY 'Unsuccessful query.'
         END-IF
      ELSE
         DISPLAY 'Call to CEEQRYL failed. ' Msg-No
      END-IF.

      STOP RUN.
  END PROGRAM COBSETL.
```

Figure 104. Calls to CEESETL and CEEQRYL in COBOL (Part 2 of 2)

Calls to CEESETL and CEEQRYL in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMSETL                           */
 /****************************************************/
 /* Example for callable service CEESETL              */
 /* Function: Set all global locale environment       */
 /*  categories to country. Query one category.       */
 /****************************************************/

 PLISETL: PROC OPTIONS(MAIN);

 %INCLUDE CEEIBMAW; /* ENTRY defs, macro defs         */
 %INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants    */
 %INCLUDE CEEIBMLC; /* Locale category constants      */

 /* CEESETL service call arguments */
 DCL LOCALE_NAME CHAR(14) VARYING;

 /* CEEQRYL service call arguments */
 DCL LOCALE_NAME_TIME CHAR(256) VARYING;

 DCL 01 FC            FEEDBACK;

   /* init locale name with IBM default for Sweden  */
   LOCALE_NAME = 'Sv_SE.IBM-1047';

   /* use LC_ALL category const from CEEIBMLC */
   CALL CEESETL ( LOCALE_NAME, LC_ALL, FC );

   /* FBCHECK macro used (defined in CEEIBMCT) */
   IF FBCHECK( FC, CEE2KE ) THEN
     DO;  /* invalid locale name */
       DISPLAY ( 'Locale LC_ALL Call '||FC.MsgNo );
       STOP;
     END;

   /* retrieve active locale for LC_TIME category */
   /* use LC_TIME category const from CEEIBMLC */
   CALL CEEQRYL ( LC_TIME, LOCALE_NAME_TIME, FC );

   IF FBCHECK( FC, CEE000 ) THEN
     DO;  /* successful query, check category name */
       IF LOCALE_NAME_TIME ¬= LOCALE_NAME THEN
         DO;
           DISPLAY ( 'Invalid LOCALE_NAME_TIME ' );
           STOP;
         END;
       ELSE
         DO;
           PUT SKIP LIST('Successful query LC_TIME',
                         LOCALE_NAME_TIME);
         END;
     END;
   ELSE
     DO;
       DISPLAY ( 'LC_TIME Category Call '||FC.MsgNo );
       STOP;
     END;

 END PLISETL;
```

Figure 105. Calls to CEESETL and CEEQRYL in PL/I

Examples Illustrating Calls to CEEQRYL and CEESTXF

The following examples illustrate calls to CEEQRYL to retrieve the locale name, and to CEESTXF to translate a string into its collation weights.

Calls to CEEQRYL and CEESTXF in COBOL

```
 CBL LIB,APOST,RMODE(ANY)
*Module/File Name: IGZTSTXF
*************************************************
* Example for callable service CEESTXF          *
*  COBSTXF - Query current collate category and *
*             build input string as function of *
*             locale name.                      *
*            Translate string as function of    *
*             locale.                           *
*************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  COBSTXF.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
* OMIT is a dummy parameter used across LE call.
 01  OMIT           COMP-2.
 01  MBS.
     02  MBS-Length  PIC S9(4) BINARY.
     02  MBS-String  PIC X(10).
 01  TXF.
     02  TXF-Length  PIC S9(4) BINARY.
     02  TXF-String  PIC X(256).
 01  Locale-Name.
     02  LN-Length  PIC S9(4) BINARY.
     02  LN-String  PIC X(256).
* Use Locale category constants
 COPY CEEIGZLC.
*
 01  MBS-Size    PIC S9(9)  BINARY  VALUE 0.
 01  TXF-Size    PIC S9(9)  BINARY  VALUE 0.
 01  FC.
     02  Condition-Token-Value.
     COPY CEEIGZCT.
         03  Case-1-Condition-ID.
             04  Severity    PIC S9(4) BINARY.
             04  Msg-No      PIC S9(4) BINARY.
         03  Case-2-Condition-ID
                   REDEFINES Case-1-Condition-ID.
             04  Class-Code  PIC S9(4) BINARY.
             04  Cause-Code  PIC S9(4) BINARY.
         03  Case-Sev-Ctl    PIC X.
         03  Facility-ID     PIC XXX.
     02  I-S-Info            PIC S9(9) BINARY.
*
 PROCEDURE DIVISION.
*************************************************
*  Call CEEQRYL to retrieve locale name
*************************************************
     CALL 'CEEQRYL' USING LC-COLLATE,
                          Locale-Name, FC.
```

Figure 106. Calls to CEEQRYL and CEESTXF in COBOL (Part 1 of 2)

```
*************************************************
*  Check feedback code and set input string
*************************************************
     IF Severity = 0
        IF LN-String (1:LN-Length) =
              'Sv-SE.IBM-1047'
           MOVE 10 TO MBS-Length
           MOVE 10 TO MBS-Size
           MOVE '7,123,456.'
                   TO MBS-String (1:MBS-Length)
        ELSE
           MOVE 7 TO MBS-Length
           MOVE 7 TO MBS-Size
           MOVE '8765432'
                  TO MBS-String (1:MBS-Length)
        END-IF
     ELSE
        DISPLAY 'Call to CEEQRYL failed. ' Msg-No
        STOP RUN
     END-IF.

     MOVE SPACES TO TXF-String.
     MOVE 0 to TXF-Length.

*************************************************
*  Call CEESTXF to translate the string
*************************************************
     CALL 'CEESTXF' USING OMIT, MBS, MBS-Size,
                          TXF, TXF-Size, FC.

*************************************************
*  Check feedback code and return length
*************************************************
     IF Severity = 0
        IF TXF-Length > 0
           DISPLAY 'Translated string is '
                   TXF-String
        ELSE
           DISPLAY 'String not translated.'
        END-IF
     ELSE
        DISPLAY 'Call to CEESTXF failed. ' Msg-No
     END-IF.

     STOP RUN.
 END PROGRAM COBSTXF.
```

Figure 106. Calls to CEEQRYL and CEESTXF in COBOL (Part 2 of 2)

Calls to CEEQRYL and CEESTXF in PL/I

```
*PROCESS MACRO;
 /*Module/File Name: IBMSTXF                                */
 /***************************************************/
 /* Example for callable service CEESTXF               */
 /* Function: Query current collate category and     */
 /*   build input string as function of locale name. */
 /*   Translate string as function of locale.        */
 /***************************************************/
 PLISTXF: PROC OPTIONS(MAIN);

 %INCLUDE CEEIBMAW; /* ENTRY defs, macro defs        */
 %INCLUDE CEEIBMCT; /* FBCHECK macro, FB constants   */
 %INCLUDE CEEIBMLC; /* Locale category constants     */

 /* CEESTXF service call arguments */
 DCL MBSTRING  CHAR(10) VARYING;   /* input string   */
 DCL MBNUMBER  BIN FIXED(31);      /* input length   */
 DCL TXFSTRING CHAR(256) VARYING;  /* output string  */
 DCL TXFLENGTH BIN FIXED(31);      /* output length  */

 /* CEEQRYL service call arguments */
 DCL LOCALE_NAME_COLLATE CHAR(256) VARYING;

 DCL 01 FC     FEEDBACK;

   /* retrieve active locale for collate category */
   /* Use LC_COLLATE category const from CEEIBMLC */
   CALL CEEQRYL ( LC_COLLATE, LOCALE_NAME_COLLATE, FC);

   /* FBCHECK macro used (defined in CEEIBMCT) */
   IF FBCHECK( FC, CEE000 ) THEN
     DO; /* successful query, set string for CEESTXF */
       IF LOCALE_NAME_COLLATE = 'Sv_SE.IBM-1047' THEN
           MBSTRING = '7,123,456.';
       ELSE
           MBSTRING = '8765432';

       MBNUMBER = LENGTH(MBSTRING);
     END;
   ELSE
     DO;
       DISPLAY ( 'Locale LC_COLLATE  '||FC.MsgNo );
       STOP;
     END;

   TXFSTRING = ';
   CALL CEESTXF ( *, MBSTRING, MBNUMBER,
                  TXFSTRING, TXFLENGTH, FC );

   IF FBCHECK( FC, CEE000 ) THEN
     DO; /* successful call, use transformed length */
       IF TXFLENGTH >0 THEN
         DO;
           PUT SKIP LIST( 'Transformed string is '||
                   SUBSTR(TXFSTRING,1, TXFLENGTH) );
         END;
     END;
   ELSE
     DO;
       IF FBCHECK( FC, CEE3TF ) THEN
         DO;
           DISPLAY ( 'Zero length input string' );
         END;
     END;

 END PLISTXF;
```

Figure 107. Calls to CEEQRYL and CEESTXF in PL/I

Chapter 19. General Callable Services

This chapter describes the set of LE/VSE callable services that provide general services. The general callable services are a set of callable services that are not directly related to a specific LE/VSE function.

Related Services

Callable Services

CEE5DMP Generates a dump of the LE/VSE run-time environment and member language libraries

CEE5PRM Passes to the calling routine the argument string with a length of up to 80 characters, that was specified at invocation of the program

CEE5PRML CEE5PRML can be used instead of CEE5PRM. It passes to the calling routine an argument string with a length of up to 300 characters, that was specified at invocation of the program.

CEE5TSTG CEE5TSTG tests for the access that is available to a specified storage address.

CEE5USR Sets or queries one of two 4-byte fields known as the user area fields

CEEGPID Retrieves LE/VSE version and platform ID

CEERAN0 Generates a sequence of uniform pseudorandom numbers between 0.0 and 1.0

CEETEST Invokes a debug tool, such as Debug Tool for VSE/ESA

Refer to the *LE/VSE Programming Reference* for detailed information about these callable services.

CEE5DMP Callable Service

CEE5DMP generates a dump of LE/VSE and the member language libraries. Sections of the dump are selectively included, depending on options specified with the *options* parameter. Output from CEE5DMP is written to the default filename CEEDUMP, unless you specify the filename of another file by using the FNAME option of CEE5DMP. The call to CEE5DMP does not cause your application to terminate. For an example of a dump and a description of the LE/VSE dump service, see *LE/VSE Debugging Guide and Run-Time Messages*.

CEE5DMP can be called by your application when you want:
- A trace of calls so you can see the order in which applications were called
- A dump of storage and control blocks
- The status of files to determine whether a file is open or closed, and to see the buffer contents of the file

© Copyright IBM Corp. 1991, 2005

CEE5PRM Callable Service

CEE5PRM queries and returns to the calling program the parameter string specified at invocation of the program. The parameter string is returned in an 80-byte fixed-length string. Only program arguments are returned, not run-time options. If the parameter string is longer than 80 characters, it is truncated. If it is shorter than 80 characters, the returned string is padded with blanks. If no program arguments are provided at invocation, the returned string is blank.

CEE5PRML Callable Service

CEE5PRML can be used instead of CEE5PRM. It queries and returns to the calling program the parameter string specified at invocation of the program. The parameter string is returned in a 300-byte fixed-length string. Only program arguments are returned, not run-time options. If the parameter string is shorter than 300 characters, the returned string is padded with blanks. If no program arguments are provided at invocation, the returned string is blank.

CEE5TSTG Callable Service

CEE5TSTG allows a programmer to test for the access that is available to a specified storage address. The service returns a feedback token indicating what access is available to the supplied storage address. This can be either:
- No access permitted in current execution key.
- Read-only access.
- Update access.
- All access permitted.

CEE5USR Callable Service

CEE5USR sets or queries one of two 4-byte fields known as the user area fields. The user area fields are associated with an enclave and are maintained on an enclave basis. A user area might be used by vendor or application programs to store a pointer to a global data area or to keep a recursion counter.

The LE/VSE user area fields should not be confused with the PL/I user area. The PL/I user area is a 4-byte field in the PL/I task communication area (TCA) and can only be accessed through assembler language.

CEEGPID Callable Service

CEEGPID retrieves the LE/VSE version ID and the platform ID.

The version ID returned by CEEGPID can be tested to determine if you can use new or extended functions that are available in a particular release of LE/VSE or on a particular platform. For example, the CEE5CIB and CEECBLDY callable services and the locale callable services are new in Release 4. Before using any of these new functions, you can test the LE/VSE version to make sure you are running on the release of LE/VSE that supports them.

You can also use CEEGPID if you are writing an application that you plan to run in the VSE, z/OS, or VM, language environments. With CEEGPID, you can check the platform ID at run-time to determine the platform you are running on. Certain functions available in the z/OS or VM language environments are not available with LE/VSE. Also, platform-specific LE/VSE callable services have different

names in the z/OS and VM language environments. For example, the LE/VSE dump callable service CEE5DMP, is called CEE3DMP in the z/OS and VM language environments.

Note: Object programs generated by LE/VSE-conforming HLL compilers are not necessarily portable between operating environments. For more information about object program portability, see your HLL publications.

CEERAN0 Callable Service

CEERAN0 generates a sequence of uniform pseudorandom numbers between 0.0 and 1.0 using the multiplicative congruential method with a user-specified seed. The numbers generated are pseudorandom in that the same numbers are generated if the same seed key is used.

CEETEST Callable Service

CEETEST invokes a debug tool, such as the Debug Tool for VSE/ESA. You can use a debug tool to monitor, trace, and interact with your application while it runs. The invocation is dynamic; the debug tool starts when errors are encountered, so you do not have to run your application under an active debug tool.

For more information about Debug Tool for VSE/ESA, see *Debug Tool for VSE/ESA User's Guide and Reference*

Examples of Using Basic Callable Services

If you plan to use an LE/VSE callable service, you must code a call to the service in your source code, then recompile your source under the latest LE/VSE-conforming version of the language you are writing in. The standard call to an LE/VSE service is different in each language, but does not differ across operating systems.

The following examples illustrate how the CEEFMDT callable service is called in C, PL/I, and COBOL. CEEFMDT sets the default date and time formats for a specified country. In the examples, `country` is a 2-character fixed-length string representing an LE/VSE-defined country code. Picture string (`pic_str` or `PICSTR`) is a character string, containing the default date and time for the country, that is returned by CEEFMDT. A feedback code (`fc`) returned from the service is checked to determine if the service completed correctly.

```
/*Module/File Name:  EDCSTRT  */
/***************************************************************/
/*                                                             */
/*  Function:  CEEFMDT - Obtain default date and time format    */
/*                                                             */
/***************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

int main(void) {

   _FEEDBACK fc;
   _CHAR2 country;
   _CHAR80 date_pic;

   /* get the default date and time format for Canada */
   memcpy(country,"CA",2);
   CEEFMDT(country,date_pic,&fc);
   if ( _FBCHECK ( fc , CEE000 ) != 0 ) {
      printf("CEEFMDT failed with message number %d\n",
             fc.tok_msgno);
      exit(2999);
   }
   /* print out the default date and time format */
   printf("%.80s\n",date_pic);
}
```

Figure 108. C Routine with a Call to CEEFMDT

```
CBL LIB,APOST
      *Module/File Name: IGZTSTRT
      *************************************************
      **                                             **
      ** CBLFMDT - Call CEEFMDT to obtain default    **
      **           date & time format                **
      **                                             **
      *************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CBLFMDT.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  COUNTRY                 PIC X(2).
       01  PICSTR                  PIC X(80).
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) BINARY.

       PROCEDURE DIVISION.
       PARA-CBLFMDT.
      **************************************************
      ** Specify country code for the US              **
      **************************************************
           MOVE 'US' TO COUNTRY.
      **************************************************
      ** Call CEEFMDT to return the default date and  **
      **     time format for the US                   **
      **************************************************
           CALL 'CEEFMDT' USING COUNTRY, PICSTR, FC.

      **************************************************
      ** If CEEFMDT runs successfully, display result.**
      **************************************************
           IF CEE000 of FC THEN
               DISPLAY 'The default date and time '
                   'format for the US is: ' PICSTR
           ELSE
               DISPLAY 'CEEFMDT failed with msg '
                   Msg-No of FC UPON CONSOLE
               STOP RUN
           END-IF.

           GOBACK.
```

Figure 109. COBOL Program with a Call to CEEFMDT

```
*PROCESS MACRO;
 /*Module/File Name: IBMSTRT
 /*******************************************/
 /**                                       **/
 /** Function: CEEFMDT - obtain default    **/
 /**                    date & time format **/
 /**                                       **/
 /*******************************************/

 PLIFMDT: PROC OPTIONS(MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL COUNTRY CHARACTER ( 2 );
    DCL PICSTR  CHAR(80);
    DCL 01 FC   FEEDBACK;

    COUNTRY = 'US'; /* Specify country code for     */
                    /* the United States            */

    /* Call CEEFMDT to get default date format      */
    /*    for the US                                */
    CALL CEEFMDT ( COUNTRY , PICSTR , FC );

    /* Print default date format for the US         */
    IF  FBCHECK( FC, CEE000)  THEN  DO;
       PUT SKIP LIST( 'The default date and time '
          || 'format for the US is ' || PICSTR );
       END;
    ELSE  DO;
       DISPLAY( 'CEEFMDT failed with msg '
          || FC.MsgNo );
       STOP;
       END;


 END PLIFMDT;
```

Figure 110. PL/I Routine with a Call to CEEFMDT

See *LE/VSE Programming Reference* for detailed instructions on how to call LE/VSE services and for more information about the CEEFMDT callable service.

Chapter 20. Math Services

This chapter introduces LE/VSE math services and describes the call interface to the math services.

Understanding the Basics

LE/VSE math services provide standard math computations and can be called from LE/VSE-conforming languages or from Language Environment-conforming assembler routines.

You can invoke LE/VSE math services by using the call interface (defined below) or by using the C, COBOL, or PL/I built-in math functions specific to the HLL used in your application. For example, your COBOL program can continue to use the built-in SIN function without having to be recoded to use the CEExSIN call interface.

© Copyright IBM Corp. 1991, 2005

Math Services

CEESxABS	Absolute value
CEESxACS	Arccosine
CEESxASN	Arcsine
CEESxATH	Hyperbolic arctangent
CEESxATN	Arctangent
CEESxAT2	Arctangent of two arguments
CEESxCJG	Conjugate complex
CEESxCOS	Cosine
CEESxCSH	Hyperbolic cosine
CEESxCTN	Cotangent
CEESxDIM	Positive difference
CEESxDVD	Division
CEESxERC	Error function complement
CEESxERF	Error function
CEESxEXP	Exponential (base e)
CEESxGMA	Gamma function
CEESxIMG	Imaginary part of a complex
CEESxINT	Truncation
CEESxLGM	Log gamma function
CEESxLG1	Logarithm base 10
CEESxLG2	Logarithm base 2
CEESxLOG	Logarithm base e
CEESxMLT	Floating-point complex multiplication
CEESxMOD	Modular arithmetic
CEESxNIN	Nearest integer
CEESxNWN	Nearest whole number
CEESxSGN	Transfer of sign
CEESxSIN	Sine
CEESxSNH	Hyperbolic sine
CEESxSQT	Square root
CEESxTAN	Tangent
CEESxTNH	Hyperbolic tangent
CEESxXPx	Exponential (**)

See *LE/VSE Programming Reference* for syntax and examples of the math services.

Call Interface to Math Services

The syntax for math services has two forms, depending on how many input parameters the routine requires. The first four letters of the math services are always CEES. The fifth character is *x*, which you replace according to the parameter types listed in "Parameter Types: parm1 Type and parm2 Type." The last three letters indicate the math function performed. In these examples, the first function performed is the absolute value (ABS), and the second function is the positive difference (DIM).

One Parameter

```
►►──CEESxABS──(──parm1──,──fc──,──result──)──────────────────────────►◄
```

Two Parameters

```
►►──CEESxDIM──(──parm1──,──parm2──,──fc──,──result──)────────────────►◄
```

Parameter Types: parm1 Type and parm2 Type

The first parameter (*parm1*) is mandatory. The second parameter (*parm2*) is used only when you use a math service with two parameters. The *x* in the fifth character position of CEESx must be replaced by a parameter type for input and output. Substitute I, S, D, Q, T, E, or R for *x*:

I 32-bit binary integer
S 32-bit single floating-point number
D 64-bit double floating-point number
Q 128-bit extended floating-point number
T 32-bit single floating-complex number [5]
E 64-bit double floating-complex number [6]
R 128-bit extended floating-complex number [7]

LE/VSE math services expect normalized input. Generally, the result has the same parameter type as the input argument.

C, COBOL, and PL/I offer built-in math functions that you can also use under LE/VSE. See *LE/VSE C Run-Time Library Reference IBM COBOL for VSE/ESA Language Reference* , and *IBM PL/I for VSE/ESA Language Reference* for a description of these functions.

5. This parameter type is comprised of a 32-bit real part and a 32-bit imaginary part.

6. This parameter type is comprised of a 64-bit real part and a 64-bit imaginary part.

7. This parameter type is comprised of a 128-bit real part and a 128-bit imaginary part.

Examples of Calling Math Services

The following examples illustrate calls to the CEESSLOG math service to calculate the logarithm base e of an argument.

Calling CEESSLOG in C

```
/*Module/File Name:  EDCMATH  */
/*******************************************************/
/*                                                     */
/* This routine demonstrates calling the math service  */
/* CEESSLOG in C/370                                   */
/*******************************************************/

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
#include <ceeedcct.h>

int main (void) {

  float int1, intr;

  _FEEDBACK fc;

  int1 = 39;
  CEESSLOG(&int1,&fc,&intr);
  if ( _FBCHECK ( fc , CEE000 ) != 0 )
  {
     printf("CEESSLOG failed with message number %d\n",
            fc.tok_msgno);
     exit(2999);
  }

  printf("Log base e of %f is %f\n",int1,intr);
}
```

Figure 111. C Call to CEESSLOG—Logarithm Base e

Calling CEESSLOG in COBOL

```
CBL LIB,APOST
      *Module/File Name: IGZTMATH
      *************************************************************
      **                                                         **
      ** Demonstrates the CEESSLOG math service in COBOL.        **
      **                                                         **
      *************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. MTHSLOG.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  ARG1RS    COMP-1.
       01  RESLTRS   COMP-1.
       01  FC.
           02  Condition-Token-Value.
           COPY  CEEIGZCT.
               03  Case-1-Condition-ID.
                   04  Severity    PIC S9(4) BINARY.
                   04  Msg-No      PIC S9(4) BINARY.
               03  Case-2-Condition-ID
                         REDEFINES Case-1-Condition-ID.
                   04  Class-Code  PIC S9(4) BINARY.
                   04  Cause-Code  PIC S9(4) BINARY.
               03  Case-Sev-Ctl    PIC X.
               03  Facility-ID     PIC XXX.
           02  I-S-Info            PIC S9(9) BINARY.

       PROCEDURE DIVISION.

       PARA-MTHSLOG.

           MOVE 5.65 TO ARG1RS.
           CALL 'CEESSLOG' USING ARG1RS, FC, RESLTRS.
      ****************************************************
      ** If CEESSLOG runs successfully, display result.**
      ****************************************************
           IF  CEE000 of FC  THEN
              DISPLAY 'SLOG OF ' ARG1RS ' = ' RESLTRS
           ELSE
              DISPLAY 'CEESSLOG failed with msg '
                   Msg-No of FC UPON CONSOLE
              STOP RUN
           END-IF.

           GOBACK.
```

Figure 112. Call to CEESSLOG—Logarithm Base e in COBOL

Calling CEESSLOG in PL/I

```
*PROCESS MACRO;
  /*Module/File Name: IBMMATH            */
     /*********************************************************/
     /*                                                       */
     /* Demonstrates the CEESSLOG math service in PL/I.       */
     /*                                                       */
     /*********************************************************/

 MTHSLOG: PROC OPTIONS(MAIN);

    %INCLUDE  CEEIBMAW;
    %INCLUDE  CEEIBMCT;

    DCL 01 FC  FEEDBACK;
    DCL ARG1   FLOAT4 INIT(5.65);
    DCL RESULT FLOAT4;

    CALL CEESSLOG (ARG1, FC, RESULT);
    IF  FBCHECK( FC, CEE000)  THEN
       PUT SKIP LIST( 'SLOG OF ' || ARG1 || ' is ' || RESULT );
    ELSE
       PUT SKIP LIST( 'CEESLOG failed with msg '|| FC.MsgNo );

 END MTHSLOG;
```

Figure 113. Call to CEESSLOG—Logarithm Base e in PL/I

Part 4. Using Interfaces to Other Products

Chapter 21. Compatibility with Other Products 295
Required Licensed Programs 295
Optional Licensed Programs 295

Chapter 22. Running Applications under CICS 297
Understanding the Basics 297
CICS Partition 297
CICS Transaction 297
CICS Run Unit 297
Running LE/VSE Applications under CICS . . 298
Developing an Application under CICS. 298
COBOL Coding Considerations under CICS . . 299
PL/I Coding Considerations under CICS . . . 299
Link-Edit Considerations under CICS 299
C Considerations 300
COBOL Considerations 300
PL/I Considerations 300
Specifying Run-Time Options under CICS . . . 300
Accessing DL/I Databases from CICS 302
Using Callable Services under CICS 303
DOS/VS COBOL Compatibility Considerations 303
Using Math Services in PL/I under CICS . . . 303
Coding Program Termination in PL/I under CICS 303
Storage Management 303
CICS Short-on-Storage Condition 303
PL/I Storage Considerations under CICS . . . 304
Initializing Static External Data 304
PL/I Object Program Size 304
Using CICS Storage Constructs Rather Than PL/I Language Statements 304
PL/I Storage Classes 304
Using Storage Built-In Functions 305
Condition Handling under CICS 305
PL/I Considerations for Using the CICS HANDLE ABEND Command 305
Effect of the CICS HANDLE ABEND Command 306
Effect of CICS HANDLE CONDITION and CICS HANDLE AID 306
Restrictions on User-written Condition Handlers 306
COBOL Considerations 306
CICS Transaction Abend Codes 307
Using the CBLPSHPOP Run-Time Option under CICS 307
Restrictions on Assembler User Exits under CICS 307
PL/I Considerations 307
Ensuring Transaction Rollback under CICS . . 307
Run-Time Output under CICS 308
Message Handling under CICS 308
PL/I SYSPRINT 309
Dump Services under CICS. 309
PL/I Considerations 309
Support for Calls within the Same HLL under CICS 309
C 309
COBOL 309
COBOL/VSE 309
VS COBOL II 310
DOS/VS COBOL 310
PL/I 310

Chapter 23. Running Applications with DB2 . . 311
Understanding the Basics 311
LE/VSE Support for DB2 Applications 311
Specifying Run-Time Options with DB2 . . . 311
Condition Handling under DB2 311

Chapter 24. Running Applications with DL/I . . 313
Understanding the Basics 313
Using the Interface between LE/VSE and DL/I . . 313
CICS Considerations 313
C Considerations 313
PL/I Considerations 314
Specifying Run-Time Options with DL/I . . . 314
Condition Handling with DL/I 314

This section describes how to link and run applications under CICS, and with DB2 and DL/I.

© Copyright IBM Corp. 1991, 2005

Chapter 21. Compatibility with Other Products

This chapter lists products that are compatible with LE/VSE. From z/VSE 3.1 onwards, LE/VSE is included in the VSE Central Functions.

Required Licensed Programs

The licensed programs in Table 43 are required to customize LE/VSE, or to run LE/VSE applications.

Table 43. Required Licensed Programs for LE/VSE

Licensed Program Name	Program Number
High Level Assembler for VSE	5696-234

Optional Licensed Programs

The licensed compiler programs listed in Table 44, with or without the Debug Tool feature, can optionally be used to generate LE/VSE applications.

Table 44. Optional Licensed Compiler Programs for LE/VSE

Licensed Program Name	Program Number
C/VSE	5686-A01
COBOL/VSE	5686-068
PL/I VSE	5686-069

The licensed programs listed in Table 45 can optionally be used with LE/VSE.

Table 45. Other Licensed Programs for LE/VSE

Licensed Program Name	Program Number
CICS/VSE	5686-026
CICS Transaction Server for VSE/ESA	5648-054
DB2 V7 Server for VSE (including QMF)	5697-F42
DFSORT/VSE	5746-SM3
DL/I DOS/VS (Release 10)	5746-XX1
DL/I VSE (Release 11)	5746-XX1

Note: LE/VSE is not supported in VSE/ICCF interactive partitions.

© Copyright IBM Corp. 1991, 2005

Chapter 22. Running Applications under CICS

LE/VSE provides support that, when used in conjunction with facilities provided by the Customer Information Control System (CICS) product permits you to write applications in high-level languages and run them in a CICS environment. LE/VSE supports CICS/VSE Version 2 Release 3, and CICS Transaction Server for VSE Version 1 Release 1.

You can code an application that runs in a CICS environment by using any LE/VSE-conforming HLL. This chapter describes special features and considerations that apply to LE/VSE-conforming applications running in a CICS environment.

Note: Wherever **CICS** is used here, it covers both CICS/VSE 2.3 and the CICS Transaction Server for VSE/ESA, *unless stated otherwise*.

Understanding the Basics

Before discussing how to develop and run LE/VSE-conforming applications in a CICS environment, it is important to map familiar CICS terminology to the terminology used in the LE/VSE program model described in Chapter 9, "Program Management Model," on page 75.

CICS Partition

A CICS partition is a fixed-size subdivision of main storage that is initialized and used by CICS. Initialization of a partition creates a common environment for all CICS transactions running in that environment. There are no unique LE/VSE services that can be applied at a partition level.

CICS Transaction

A CICS transaction is initiated by a single request, usually from a terminal. A CICS transaction is equivalent to an LE/VSE process. An LE/VSE process consists of one or more enclaves that carry out the needed processing when they are run. When a CICS transaction is initiated, the first LE/VSE thread is triggered within the first enclave in the LE/VSE process.

For example, the insertion of a bank card into an ATM might trigger an LE/VSE process (CICS transaction) consisting of one or more enclaves (CICS run units) to read the information on the card. After an ATM reads a bank card, the validation of the information on the card might be performed by one enclave, processing the user's personal id number might be performed by another enclave, processing a user request by another, and dispensing the cash by a final enclave.

CICS Run Unit

A CICS run unit consists of a bound set of one or more phases that can be loaded by the CICS program loader. Run units are equivalent to LE/VSE enclaves. Each enclave has its own entry in the CICS processing program table (PPT). (The PPT can be updated using the CICS DFHPPT macro or the CICS system definition (CSD) file.) Under CICS, it is possible for a single enclave to have multiple separately link-edited phases with separate entries in the PPT. Each enclave has its own heap storage and other LE/VSE resources associated with it.

© Copyright IBM Corp. 1991, 2005

An enclave is invoked when an LE/VSE process (CICS transaction) is triggered or when it is passed control from another enclave using the EXEC CICS LINK or EXEC CICS XCTL commands. For details on using EXEC CICS LINK or EXEC CICS XCTL commands, see "Creating Child Enclaves Using EXEC CICS LINK or EXEC CICS XCTL" on page 394.

Running LE/VSE Applications under CICS

The following steps describe basic application execution under CICS:

1. An event, generally the receipt of an input message containing a transaction ID code, triggers an LE/VSE process (CICS transaction).
2. CICS looks up the transaction ID code in the program control table (PCT) and gets the name of the enclave (or the first enclave) to execute the process.
3. CICS defines the process (transaction) as a work item that is dispatched by the CICS task dispatcher.
4. Once the process is defined, CICS looks up the identity of the enclave required to perform the task in the PPT. The PPT contains information about the enclave such as its language, whether it is in storage, and if in storage, its use count and entry point address.
5. CICS calls the LE/VSE-CICS run-time level interface to initialize the process-related portions of the run-time environment.
6. If the enclave does not perform all the processing associated with the process, the enclave might pass control to another enclave through a language call or through the EXEC CICS LINK or EXEC CICS XCTL commands.
7. When the process is complete, CICS calls the LE/VSE-CICS run-time level interface to terminate the process-related portions of the run-time environment.

Developing an Application under CICS

Certain coding restrictions apply when you develop an application to run under CICS. Examples are:

- Input/output restrictions—CICS provides its own I/O facilities using various EXEC CICS commands.
- Multitasking—CICS has its own multitasking capability.

After you code your application, you must run it through a *CICS translator*. The translator accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source. The CICS translator runs in a separate job step. The job step sequence for preparing and running an application under CICS is:

1. Code
2. Translate
3. Compile
4. Prelink (C only)
5. Link-Edit
6. Run

C coding restrictions are discussed in *LE/VSE C Run-Time Programming Guide* and COBOL restrictions are discussed in *IBM COBOL for VSE/ESA Programming Guide* Examples of PL/I coding restrictions under CICS are discussed in "PL/I Coding Considerations under CICS" on page 299.

For more information on developing an application under CICS see *CICS Transaction Server for VSE/ESA Application Programming Guide* listed in "Where to Find More Information" on page xxi.

COBOL Coding Considerations under CICS

When coding VS COBOL II or COBOL/VSE programs to run under CICS, you must specify either the XOPTS(ANSI85) or XOPTS(COBOL2) translator option when you translate the program. If you do not specify one of these options, your application will abend.

For COBOL/VSE applications that are translated using the CICS TS for VSE/ESA translator, the preferred option setting is XOPTS(COBOL3).

PL/I Coding Considerations under CICS

When coding PL/I routines to run under CICS, consider the following:

- Built-in subroutines—There are some restrictions on the use of PL/I's built-in subroutines:
  - You cannot use the PLISRT*x* interfaces, PLICKPT or PLICANC.
  - You can use PLIRETC and PLIRETV to communicate between user-written routines that are link-edited together, but not to communicate with CICS. See "Managing Return Codes in LE/VSE" on page 68 for details.
- Debugging facilities— Support has been added to allow the CICS transaction to be debugged using a debug tool, such as Debug Tool for VSE/ESA. To prepare your program to a debug tool, you must compile with the TEST option. For more information on debugging under LE/VSE, see *LE/VSE Debugging Guide and Run-Time Messages*.
- Language I/O facilities—You can use only a subset of PL/I I/O facilities under CICS.
  - OPEN/CLOSE can be used, but only for the SYSPRINT file, and only if the SYSPRINT file is declared (implicitly or explicitly) with the EXTERNAL PRINT attributes
  - You can use stream output only to the SYSPRINT file (with EXTERNAL PRINT attributes). For performance reasons, you should use stream output under CICS only when debugging your applications.

  Those PL/I I/O facilities that you cannot use under CICS are:
  - Record I/O statements
  - Stream input
  - DISPLAY statement
  - DELAY statement
  - STOP statement
  - WAIT statement
  - PL/I I/O-related conditions such as RECORD, TRANSMIT, ENDFILE, and KEY are not raised under CICS, because I/O is not performed using PL/I files (except SYSPRINT) and I/O statements. CICS file-handling facilities are used instead. If CICS detects an I/O condition during the processing of your commands, CICS deals with the condition in the way defined in the CICS manuals.

Link-Edit Considerations under CICS

You can link-edit LE/VSE-conforming applications that are to be run under CICS just as if they were batch applications. If your C, COBOL, or PL/I application uses EXEC CICS commands, however, you must also link-edit the EXEC CICS interface

stub, DFHELII, with your application. To be link-edited with your application, DFHELII must be available in the object sublibrary chain. See *CICS Transaction Server for VSE/ESA System Definition and Operations Guide* for more information.

C Considerations

C applications must be link-edited AMODE(31), RMODE(ANY), as shown in “C AMODE/RMODE Considerations” on page 9.

COBOL Considerations

DFHELII is compatible with the DFHECI stub provided for COBOL programs.

Although DFHECI is still supported under LE/VSE, DFHELII offers some advantages. Whereas the old COBOL stub had to be link-edited at the top of your application, DFHELII can be linked anywhere in the application. You also have the capability of linking ILC applications with a single stub rather than with multiple stubs.

PL/I Considerations

You no longer need to include DFHPL1I. DFHPL1I is not supported by PL/I VSE. You should change DFHPL1I to DFHELII in your link-edit jobs.

You no longer need to include PLISHRE, the generalized shared library interface. PLISHRE is not supported by PL/I VSE and should be removed from your link-edit jobs.

Specifying Run-Time Options under CICS

Under CICS, you cannot pass run-time options as parameters when the application is invoked. However, you can specify run-time options for your application using one of the following methods:

- As default options established in CEECOPT during the installation of LE/VSE (see page 36 for more information about CEECOPT)
- As application defaults established in CEEUOPT (see page 36 and “CEEXOPT Invocation Syntax” on page 40 for details)
- In the user exit (see “CEEBXITA Assembler User Exit Interface” on page 323 for details on how to do this)
- In C applications, as options specified using `#pragma runopts` (see page 36 for the proper syntax)
- In PL/I applications, as default options established in CEEUOPT using the PLIXOPT string (see page 37 for the proper syntax)

Some run-time options have different defaults and exhibit slightly different behavior while executing under CICS. The options that differ are listed in Table 46 on page 301.

Table 46. Run-Time Option Behavior under CICS

Option	Description
ABPERC	ABPERC is ignored under CICS.
AIXBLD	AIXBLD is ignored under CICS.
ALL31	ALL31(ON) is the IBM-supplied default under CICS.
ANYHEAP	ANYHEAP=((4K,4080,ANYWHERE,FREE),OVR) is the IBM-supplied default under CICS. Both the initial size and the increment size are rounded up to the nearest multiple of 8 bytes. The minimum is 4K for initial size, and 4080 bytes for increment size. Under CICS/VSE 2.3, if ANYHEAP(,,BELOW) is in effect, the maximum initial and increment size for ANYHEAP is 65,504 bytes. If ANYHEAP(,,ANYWHERE) is in effect, the maximum initial and increment size for ANYHEAP is 1 gigabyte (1024M). If you specify the ANYHEAP run-time option in CEEUOPT, the following default values are used for omitted suboptions: • init_size is 32K • incr_size is 16K
ARGPARSE	ARGPARSE is ignored under CICS.
BELOWHEAP	BELOWHEAP=((4K,4080,FREE),OVR) is the IBM-supplied default under CICS. Both the initial size and the increment size are rounded to the nearest multiple of 8 bytes. The minimum is 4K for initial size, and 4080 bytes for increment size. The maximum initial and increment size for BELOWHEAP under CICS/VSE 2.3 is 65,504 bytes. If you specify the BELOWHEAP run-time option in CEEUOPT, the following default values are used for omitted suboptions: • init_size is 32K
ENV	ENV is ignored under CICS.
EXECOPS	EXECOPS is ignored under CICS.
HEAP	HEAP=((4K,4080,ANYWHERE,KEEP,4080,4K),OVR) is the IBM-supplied default under CICS. Both the initial HEAP allocation and HEAP increments are rounded to the next higher multiple of 8 bytes. The minimum is 4K for initial size, and 4080 bytes for increment size. Under CICS/VSE 2.3, if HEAP(,,BELOW) is in effect, the maximum size of a heap segment is 65,504 bytes. If too large a value is specified, the application fails at the first attempt to allocate heap storage. If HEAP(,,ANYWHERE) is in effect, the maximum size of a heap segment is 1 gigabyte (1024M). These restrictions are subject to change from one release of CICS to another. If you specify the HEAP run-time option in CEEUOPT, the following default values are used for omitted suboptions: • init_size is 64K
LIBSTACK	LIBSTACK=((4K,4080,FREE),OVR) is the IBM-supplied default under CICS. Both the initial and increment sizes are rounded up to the next multiple of 8 bytes. The minimum is 4K for initial size, and 4080 bytes for increment size. Under CICS, the maximum initial and increment size for LIBSTACK is 65,504 bytes. If you specify the LIBSTACK run-time option in CEEUOPT, the following default values are used for omitted suboptions: • init_size is 32K • incr_size is 16K

Table 46. Run-Time Option Behavior under CICS (continued)

Option	Description
MSGFILE	MSGFILE=((CESE),OVR) is the IBM-supplied default under CICS. This means, the MSGFILE option defaults to the CESE transient data queue. Specification of a different transient data queue for MSGFILE is possible. However, it is the users responsibility to ensure that this transient data queue is available in the CICS system. The CESE transient data queue must always be available either as TYPE=INDIRECT or TYPE=EXTRA in the CICS DCT definition. The MSGFILE destination name under CICS must not exceed 4 characters in length. Truncation will occur on the MSGFILE destination if the name used is greater than 4 characters in length. See "Run-Time Output under CICS" on page 308 for further information.
PLIST	PLIST is ignored under CICS.
REDIR	REDIR is ignored under CICS.
RTEREUS	RTEREUS is ignored under CICS.
STACK	STACK=((4K,4080,ANYWHERE,KEEP),OVR) is the IBM-supplied default under CICS. The maximum initial and increment size for STACK below 16MB is 65,504 bytes. The maximum initial and increment size for STACK above 16MB is 1 gigabyte (1024M). This restriction is subject to change from one release of CICS to another. Both the initial size and the increment size are rounded up to the nearest multiple of 8 bytes. The initial size minimum is 4K, the increment size minimum is 4080 bytes. **Note:** LE/VSE uses the STACK initial size as specified in the installation defaults or programmer's defaults. LE/VSE does not use the STACK initial size if the option is specified or modified in the assembler user exit. If you want to tune your run unit execution with the STACK initial size value, you must change the value in CEEUOPT and relink-edit your application, or change the value in the `#pragma runopts` of your C routine and recompile your application. If you specify the STACK run-time option in CEEUOPT, the following default values are used for omitted suboptions: • init_size is 512K • incr_size is 512K
STORAGE	STORAGE=((00,NONE,NONE,0K),OVR) is the IBM-supplied default under CICS. The out-of-storage condition is not raised under CICS. If a reserved segment size is specified, either as a default or an override under CICS, this storage size will be allocated but never used. This results in wasted 24-bit storage. You are recommended to always use a reserve segment size of 0k under CICS. If you specify the STORAGE run-time option in CEEUOPT, the default value 8K is used if the reserve_size suboption is omitted.
TERMTHDACT	TERMTHDACT((TRACE,MSGFL,0),OVR) is the IBM-supplied default under CICS. TERMTHDACT sets the level of information that is produced when LE/VSE percolates a condition of severity 2 or greater beyond the first routine's stack frame. The UADUMP suboption has been added to TERMTHDACT as part of these enhancements to provide a comprehensive dump in the event of an abnormal termination. The LE/VSE service CEE5DMP is called for the TRACE, DUMP, and UADUMP suboptions of TERMTHDACT. The MSGFL sub-option causes all dump output to be sent to the output destination specified by the Run-Time option MSGFILE. If you specify LSTQ sub-option, all dump output will be sent to the VSE/POWER LSTQ. For further details, refer to "Appendix A: LE/VSE Run-Time Options" of the *LE/VSE Customization Guide*, SC33-6682.

Accessing DL/I Databases from CICS

Various user interfaces to DL/I databases are available under CICS. See Chapter 24, "Running Applications with DL/I," on page 313 for details.

Using Callable Services under CICS

All LE/VSE callable services are available to applications executing as CICS transactions. However, the CEEMOUT (dispatch a message) and CEE5DMP (generate dump) services differ, in that messages and dumps are sent to the transient data queue specified in the MSGFILE run-time option. See *LE/VSE Programming Reference* for descriptions of these services.

See *LE/VSE Writing Interlanguage Communication Applications* for ILC examples that make a call to CEEMOUT.

DOS/VS COBOL Compatibility Considerations

LE/VSE provides a set of compatibility library routines that permit you to run DOS/VS COBOL applications under CICS in compatibility mode. When you run a DOS/VS COBOL application on CICS, the environment that is established for a run unit by the compatibility library routines supports only DOS/VS COBOL. This compatibility library does not contain many of the services normally offered under LE/VSE. LE/VSE run-time options and callable services, for example, are not supported.

Using Math Services in PL/I under CICS

PL/I saves and restores floating-point registers where necessary. PLIDUMP can print these registers (see *LE/VSE Debugging Guide and Run-Time Messages* for more information about PLIDUMP).

Floating-point overflow and underflow can be handled in OVERFLOW and UNDERFLOW ON-units. The program mask is set appropriately for the levels of CICS and PL/I used.

Coding Program Termination in PL/I under CICS

You can terminate a PL/I routine running under CICS by using PL/I constructs or CICS statements such as EXEC CICS RETURN, EXEC CICS SEND PAGE RELEASE, EXEC CICS XCTL, or EXEC CICS ABEND. When the routine terminates, the following occurs:

1. If you requested a storage report using the RPTSTG run-time option, the report is written to the transient data queue specified in the MSGFILE run-time option (described in "Run-Time Output under CICS" on page 308).
2. If the MSGFILE destination is still open, it will be closed.
3. All storage acquired by PL/I is freed before control returns to CICS, except for the stack.

Storage Management

Applications can allocate and free storage explicitly through language facilities, CICS facilities (EXEC CICS GETMAIN and FREEMAIN commands, see *CICS Transaction Server for VSE/ESA Application Programming Reference* for more information), or the LE/VSE storage management callable services.

If you do not explicitly free storage that was allocated through language facilities or LE/VSE callable services, the storage is freed at enclave termination.

CICS Short-on-Storage Condition

The CICS short-on-storage condition might be raised under LE/VSE if functions in your application attempt to acquire storage by using language facilities and not

enough storage is available to satisfy the request. CICS places the transaction on a queue until the storage request can be satisfied. If CICS cannot get enough storage in a reasonable amount of time to satisfy the request, then the transaction that issued the storage request is terminated by CICS with abend code AKCP.

PL/I Storage Considerations under CICS

Special storage considerations for running PL/I applications under CICS are described in the following sections.

Initializing Static External Data

You must initialize static external data under CICS because CICS cannot handle common CSECTs.

PL/I Object Program Size

The maximum program size allowed for an RMODE(24) program is 512KB. The maximum program size allowed for an RMODE(ANY) program in the XA environment is 16MB (although this is not recommended).

Using CICS Storage Constructs Rather Than PL/I Language Statements

In the case when a PL/I routine (routine A, for example) issues an EXEC CICS LINK to another PL/I routine (routine B, for example), you might want to use EXEC CICS GETMAIN and FREEMAIN commands to get and free storage. This is because the scope of EXEC CICS GETMAIN is the scope of the entire task, not just a single routine. Either routine A or routine B can explicitly free the storage. Alternatively, you can choose to not explicitly free the storage in either routine, but allow the storage to be freed automatically when the task is terminated. Another advantage to using EXEC CICS GETMAIN is that if routine A terminates, the storage is still available to routine B.

When you use PL/I language statements to get and free storage, the scope of PL/I storage statements is the routine, not the task. Although routine B can alter the storage allocated by routine A by using a pointer, routine B cannot free the storage. In addition, if routine A terminates, the storage is automatically freed. Routine B can no longer access the storage.

PL/I Storage Classes

When using CICS, you should avoid altering STATIC storage. Doing so violates reentrancy and can yield unpredictable results. Instead of altering STATIC storage, you should make most or all user variables that are changed while the routine is running AUTOMATIC. Those user variables with initial values that never change should be declared STATIC INITIAL.

Although AUTOMATIC storage provides reentrancy and should suffice for most purposes, you can also allocate and free storage with the ALLOCATE and FREE statements, which you can use to allocate and free BASED and CONTROLLED variables using these statements. References you make to BASED storage are handled with the pointer set by the ALLOCATE statement. The pointer itself can be AUTOMATIC.

You can use CONTROLLED storage under CICS, because it is consistent with reentrancy.

Using PUT DATA with BASED Storage: BASED storage is used extensively in CICS transactions. You therefore need to be aware of the following restriction on PUT DATA.

In PL/I, you cannot code:

```
PUT DATA (P -> VAR);
```

If, however, VAR was declared as BASED (P), the value of the generation of VAR to which P points can be coded as:

```
PUT DATA (VAR);
```

Using Storage Built-In Functions

The STORAGE and CURRENTSTORAGE built-in functions return the length of an item to your PL/I routine. This is useful in CICS, where functions often require the length of an argument as well as its address. In particular, you can use these functions to get lengths of PL/I aggregates without having to count or compute such lengths or specify length fields in the CICS commands.

For more information about the STORAGE and CURRENTSTORAGE built-in functions, see *IBM PL/I for VSE/ESA Language Reference*

Condition Handling under CICS

The LE/VSE condition handling services described in Chapter 11, "LE/VSE Condition Handling Introduction," on page 103 and elsewhere in this book are supported under CICS, but additional considerations apply when running an application under CICS; these considerations are described in the following sections.

Condition handling in nested enclaves created by EXEC CICS LINK or EXEC CICS XCTL is discussed in "How Conditions Arising in Child Enclaves Are Handled" on page 394.

PL/I Considerations for Using the CICS HANDLE ABEND Command

The EXEC CICS HANDLE facility resembles a PL/I ON-unit with this syntax:

```
ON condition GO TO label;
```

You can code the HANDLE command wherever you would code the ON...GO TO...statement. The label to be branched to can be located in any other active block, and the condition can arise in an even later block. HANDLE terminates intervening PL/I blocks by invoking PL/I's out-of-block GO TO facilities.

Note: Because PL/I internal procedures are not active at all times, you should not use internal procedures as exit routines in HANDLE commands.

HANDLE is not semantically identical to the `ON condition GO TO label;` statement. A PL/I ON-unit disappears when the block containing it terminates; a CICS HANDLE disappears when it is explicitly overridden by another one.

A HANDLE command could specify a branch to a label in a block no longer active. Because HANDLE is implemented by forcing a PL/I out-of-block GO TO, this is equivalent to assigning a label constant to a PL/I label variable after the block containing the label constant has terminated, which is invalid. The PL/I out-of-block GO TO mechanism attempts to detect this error and raises the ERROR condition. If PL/I out-of-block GO TO fails to detect such an invalid GO TO, however, the GO TO becomes a wild branch that causes some unpredictable failure. Thus, upon return from a PL/I block that established HANDLE for a particular condition, your program should issue a resetting HANDLE for that

condition (provided, of course, that there is still some possibility of the condition arising). A PL/I ON-unit does not have to be reset.

Effect of the CICS HANDLE ABEND Command

When an application is running under CICS with LE/VSE, condition handling differs depending on whether a CICS HANDLE ABEND is active or **not** active.

When a CICS HANDLE ABEND is active, LE/VSE condition handling does not gain control for any abends or program interrupts. Any abends or program interrupts that occur while a CICS HANDLE ABEND is active cause the action defined in the CICS HANDLE ABEND to take place. The user-written condition handlers established by CEEHDLR are ignored.

When a CICS HANDLE ABEND is not active, LE/VSE condition handling does gain control for abends and program interrupts if the TRAP(ON) option is specified. Normal LE/VSE condition handling is then performed.

Effect of CICS HANDLE CONDITION and CICS HANDLE AID

LE/VSE condition handling does not alter the behavior of applications that use CICS HANDLE CONDITION or CICS HANDLE AID. The CICS CONDITION and AID conditions are raised by CICS and are handled only by CICS; LE/VSE is not involved in the handling of CICS conditions.

Restrictions on User-written Condition Handlers

The following EXEC CICS commands cannot be used within a user-written condition handler established using CEEHDLR, or within any routine called by a user-written condition handler:
- EXEC CICS ABEND
- EXEC CICS HANDLE AID
- EXEC CICS HANDLE ABEND
- EXEC CICS HANDLE CONDITION
- EXEC CICS IGNORE CONDITION
- EXEC CICS POP HANDLE
- EXEC CICS PUSH HANDLE

All other EXEC CICS commands are allowed within a user-written condition handler. However, they must be coded using the NOHANDLE option, the RESP option, or the RESP2 option. This prevents additional conditions being raised due to a CICS service failure.

COBOL Considerations

A user-written condition handler registered using the CEEHDLR service cannot be translated using the CICS translator and therefore cannot contain any EXEC CICS commands. This is because the CICS translator inserts (onto the PROCEDURE DIVISION header of the COBOL program) the arguments EXEC Interface Block (EIB) and COMMAREA, which do not match arguments passed by LE/VSE.

However, a user-written condition handler can call a subroutine to perform EXEC CICS commands. If arguments need to be passed to this subroutine, they should be preceded by two dummy arguments in the caller. The called subroutine must issue EXEC CICS ADDRESS EIB before executing any other EXEC CICS commands.

CICS Transaction Abend Codes

The same LE/VSE reserved abend codes (4000 through 4095) are used for applications running under CICS. In addition, there are special reason codes returned to CICS for severe LE/VSE conditions. These severe conditions are CICS-specific. For a detailed explanation of these reason codes, see *LE/VSE Debugging Guide and Run-Time Messages*.

Using the CBLPSHPOP Run-Time Option under CICS

This section applies to VS COBOL II and COBOL/VSE programs only.

The CBLPSHPOP run-time option controls whether the LE/VSE environment automatically issues an EXEC CICS PUSH HANDLE command during initialization and an EXEC CICS POP HANDLE command during termination whenever a VS COBOL II or COBOL/VSE subroutine is called using the COBOL CALL statement.

If your application calls COBOL subroutines under CICS, your application performance is better with CBLPSHPOP(OFF) than with CBLPSHPOP(ON). You can set CBLPSHPOP on a transaction-by-transaction basis by using CEEUOPT.

See *LE/VSE Programming Reference* for more information about CBLPSHPOP.

Restrictions on Assembler User Exits under CICS

The following EXEC CICS commands cannot be used within the assembler user exit or any routines called by the assembler user exit:

- EXEC CICS ABEND
- EXEC CICS HANDLE AID
- EXEC CICS HANDLE ABEND
- EXEC CICS HANDLE CONDITION
- EXEC CICS PUSH HANDLE
- EXEC CICS POP HANDLE
- EXEC CICS IGNORE CONDITION

All other EXEC CICS commands are allowed within the assembler user exit. However, they must be coded using the NOHANDLE option, the RESP option, or the RESP2 option. This prevents additional conditions being raised due to a CICS service failure.

See Chapter 25, “Using Run-Time User Exits,” on page 319 for a discussion of the assembler user exits available under LE/VSE.

PL/I Considerations

You can use PLIRETC to communicate with the LE/VSE assembler user exit. See “Setting and Altering User Return Codes” on page 69 for more information about PLIRETC, and Chapter 25, “Using Run-Time User Exits,” on page 319 for more information about the assembler user exit.

Ensuring Transaction Rollback under CICS

Conditions that occur while an application is executing under CICS can potentially contaminate any database currently being used by the application. It is essential that a rollback (the backing out of any updates made by the failing application) be performed before further damage to the database can occur.

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(ABEND) run-time option, or
- Make sure the assembler user exit requests an abend for unhandled conditions of severity 2 or greater.

See *LE/VSE Programming Reference* for an explanation of the ABTERMENC run-time option. See Chapter 25, “Using Run-Time User Exits,” on page 319 for more information about using assembler user exits.

Run-Time Output under CICS

LE/VSE provides the same message handling and dump services for CICS as it does for non-CICS systems. Any exceptions to this support under CICS are noted in the following sections.

Message Handling under CICS

Under CICS, the MSGFILE option defaults to the CESE transient data queue. Specification of a different transient data queue for MSGFILE is possible. However, it is the users responsibility to ensure that this transient data queue is available in the CICS system. The CESE transient data queue must always be available either as TYPE=INDIRECT or TYPE=EXTRA in the CICS DCT definition. If an option other than CESE is specified for MSGFILE and this transient data queue becomes unusable or unavailable, LE/VSE will default to the CESE transient data queue.

The MSGFILE destination name under CICS must not exceed 4 characters in length. Truncation will occur on the MSGFILE destination if the name used is greater than 4 characters in length. The supplied definition of CEEMSG in CEECDCT.A in PRD2.SCEEBASE should be used as an example for any other TYPE=SDSCI destinations being used as a Disk File destination for MSGFILE. Note that if a DISK file is being used as a final destination, you must remember to add 8 bytes to the BLKSIZE specified in your DCT definition. Any MSGFILE destination used must support a blksize of at least 175 bytes (inclusive of the 8 bytes required for LIOCS output files if DISK is used). The VSE system console is not a supported destination for MSGFILE either directly or indirectly.

Messages are prefixed by a terminal ID, a transaction ID, a date, and a timestamp before their transmission. Figure 114 illustrates this format.

ASA	Terminal ID	Transaction ID	SP	Time Stamp YYYYMMDDHHMMSS	SP	Message
1	4	4	1	14	1	132

Figure 114. Format of Messages Sent to CESE

ASA The American National Standard Code for Information Interchange (ASCII) carriage-control character (optional character).

Terminal ID
A 4-character terminal identifier.

Transaction ID
A 4-character transaction identifier.

sp A space.

Timestamp
The date and time displayed in the same format as that returned by the CEELOCT service.

Message
The message identifier and message text.

The entire message record is preceded by an ASCII control character to determine the format of the printing.

Message records are V-format.

See Chapter 15, “Using and Handling Messages,” on page 185 for a complete description of LE/VSE message handling.

PL/I SYSPRINT

PL/I SYSPRINT also uses the CESE transient data queue. For information on how to declare SYSPRINT, see *IBM PL/I for VSE/ESA Programming Guide*

Dump Services under CICS

Under CICS, the FNAME parameter of the CEE5DMP callable service is ignored. Instead of being written to a ddname specified in FNAME, dumps are instead transmitted to the CICS transient data queue named CESE.

The dump is prefixed with the same information shown in Figure 114 on page 308.

PL/I Considerations

The PLIDUMP subroutine has two additional options under CICS and some special considerations. See *LE/VSE Debugging Guide and Run-Time Messages* for more information about PLIDUMP.

Support for Calls within the Same HLL under CICS

C

EXEC CICS LINK, EXEC CICS XCTL, and calls via `fetch()` are supported under CICS. The fetched program must be defined in the PPT (either using the DFHPPT macro or the CSD). For further information, see *LE/VSE C Run-Time Programming Guide*.

COBOL

The following sections describe support for calls compiled under different versions of COBOL compilers.

COBOL/VSE

Static and dynamic calls between COBOL/VSE and VS COBOL II programs are supported as follows:

- Called programs can contain any command or facility supported by CICS for COBOL.
- If the called program has been translated by the CICS translator, calling programs must pass the EIB and COMMAREA as the first two parameters on the CALL statement.

COBOL/VSE programs can invoke or be invoked by DOS/VS COBOL programs only through CICS facilities such as EXEC CICS LINK, EXEC CICS XCTL, and EXEC CICS RETURN.

VS COBOL II

Static and dynamic calls to or from VS COBOL II and COBOL/VSE programs are supported with the same considerations previously listed for COBOL/VSE.

VS COBOL II programs can communicate with DOS/VS COBOL programs only through CICS facilities such as EXEC CICS LINK, EXEC CICS XCTL, and EXEC CICS RETURN.

DOS/VS COBOL

DOS/VS COBOL cannot call or be called by COBOL/VSE or VS COBOL II programs. Communication between a program compiled with DOS/VS COBOL and one compiled with a later version COBOL compiler is permitted only by CICS facilities such as EXEC CICS LINK, EXEC CICS XCTL, and EXEC CICS RETURN.

PL/I

Static calls are supported from PL/I. Called subroutines can invoke CICS services if the address of the EIB is passed to the subroutine properly. You can do this by setting up the address of the EIB yourself and passing it to the subroutine, or by coding the following command in the subroutine before issuing any other CICS commands.

```
EXEC CICS ADDRESS EIB(DFHEIPTR)
```

PL/I FETCH is supported under CICS in a PL/I transaction compiled

Chapter 23. Running Applications with DB2

This chapter describes LE/VSE support for DB2 applications.

Understanding the Basics

An application program requests DB2 services by using SQL statements imbedded in the program. The imbedded SQL is translated by the DB2 pre-compiler into host language statements that typically perform assignments and then call a DB2 language interface module. The same entry point for the module is called by all LE/VSE-conforming languages. DB2 processes the request and then returns to the application.

LE/VSE Support for DB2 Applications

You are not required to modify anything in your code to run an LE/VSE-conforming application with DB2. LE/VSE also supports ILC applications that use DB2 services.

For information about HLL restrictions under DB2, see the Application Programming Guide for your HLL listed in “Where to Find More Information” on page xxi. For more information about using DB2 services, see *Application Programming Guide*

Specifying Run-Time Options with DB2

With DB2, you cannot pass run-time options as parameters when the application is invoked. However, you can specify run-time options for your application using one of the following methods:

- As default options established in CEEDOPT during the installation of LE/VSE (see page 35 for more information about CEEDOPT)
- As application defaults established in CEEUOPT (see page 36 and “CEEXOPT Invocation Syntax” on page 40 for details)
- In the user exit (see “CEEBXITA Assembler User Exit Interface” on page 323 for details on how to do this)
- In C applications, as options specified using `#pragma runopts` (see page 36 for the proper syntax)
- In PL/I applications, as default options established in CEEUOPT using the PLIXOPT string (see page 37 for the proper syntax)

Condition Handling under DB2

An DB2 database can be contaminated if errors occurring in DB2 are not handled properly. For this reason, any errors occurring in DB2 must be trapped and handled by DB2. If a task terminates, DB2 can then take appropriate action depending on the nature of termination.

If you run DB2 in single-user mode, LE/VSE and DB2 keep track of calls to and returns from DB2. If a program interrupt or abend occurs when your application is running, the LE/VSE condition manager is informed whether the problem occurred in your application or in DB2. If the program interrupt or abend occurs in DB2, the LE/VSE condition handler percolates the condition back to DB2.

© Copyright IBM Corp. 1991, 2005

If you run DB2 in multiple-user mode, any errors occurring in DB2 are trapped by DB2, regardless of the LE/VSE TRAP run-time option you specify.

If a program interrupt or abend occurs in the application outside of DB2, or a software condition of severity 2 or greater is raised outside of DB2, the condition manager takes normal condition handling actions as described in Chapter 11, "LE/VSE Condition Handling Introduction," on page 103. If the condition manager gets control then you **must** do one of the following:

- Resolve the error completely so that the application can continue.
- Make sure that the application terminates abnormally by using the ABTERMENC(ABEND) run-time option to transform all abnormal terminations into operating system abends in order to cause DB2 rollbacks.
- Make sure that the application terminates abnormally by coding and providing a modified run-time assembler user exit (CEEBXITA) that transforms all abnormal terminations into operating system abends in order to cause DB2 rollbacks. The assembler user exit you provide should check the return code and reason code or the CEEAUE_ABTERM bit, and request an abend by setting the CEEAUE_ABND flag to ON, if appropriate. See "CEEBXITA Assembler User Exit Interface" on page 323 for more details about CEEBXITA user exit.

Chapter 24. Running Applications with DL/I

This chapter describes LE/VSE support for applications running with DL/I DOS/VS Version 1 Release 10 and later.

Understanding the Basics

You do not need to change any of the code in your application in order to run with DL/I DOS/VS, but there are some recommendations that you should consider to ensure proper condition handling under DL/I. This topic, together with an overview of how LE/VSE interacts with DL/I, are discussed in detail below.

For a detailed description of how to write DL/I batch and online applications, see the appropriate *DL/I DOS/VS Application Programming* book listed in "Where to Find More Information" on page xxi.

Using the Interface between LE/VSE and DL/I

LE/VSE provides a callable service, CEETDLI, that you can use to invoke DL/I. In assembler, COBOL, and PL/I, and C, you could also invoke DL/I using the following interfaces:

- In assembler, the ASMTDLI interface
- In COBOL, the CBLTDLI interface or the EXEC DLI interface
- In PL/I, the PLITDLI interface or the EXEC DLI interface
- In C, the `ctdli()` function call or the EXEC DLI interface

Under LE/VSE, each of these interfaces continues to function in its current capacity.

CEETDLI performs essentially the same functions, as these language-specific interfaces, but is language independent. Only LE/VSE-conforming application code can call CEETDLI. Calls to CEETDLI are coded in the same way as calls to the language-specific interfaces.

For information about CEETDLI, including its syntax and examples, see *LE/VSE Programming Reference*. For a complete description of all available DL/I functions and argument parameters you can specify in CEETDLI, see *DL/I DOS/VS Application Programming: CALL and RQDLI Interfaces*

The names CEETDLI, ASMTDLI, CBLTDLI, PLITDLI, and CTDLI are all interpreted to mean DL/I interfaces. If you are currently using them in any other way in your application, you must change them.

CICS Considerations

Under CICS, you can use CEETDLI as well as the existing interfaces to access DL/I databases.

C Considerations

To interface with DL/I from C, you must do the following:

- Specify the PLIST(OS), ENV(DLI), and NOEXECOPS run-time options of `#pragma runopts` in your source code. The PLIST(OS) option establishes the

© Copyright IBM Corp. 1991, 2005

correct parameter list format for DL/I. The ENV(DLI) option establishes the correct operating environment. The NOEXECOPS option specifies that run-time options cannot be specified with DL/I.

- When you use the PLIST(OS) option in `#pragma runopts`, `argc` contains 1 (one) and `argv`[0] contains NULL.

For more information about using the `#pragma runopts` preprocessor directive, see Chapter 5, "Using Run-Time Options," on page 33.

PL/I Considerations

The SYSTEM(DLI) compile option must be specified for PL/I batch applications running with DL/I. When SYSTEM(DLI) is specified, the OPTIONS(BYADDR) attribute is implied for the external PROCEDURE that also has OPTIONS(MAIN). Further, the parameters to such a MAIN procedure must be POINTERs.

Specifying Run-Time Options with DL/I

With DL/I, you cannot pass run-time options as parameters when the application is invoked. However, you can specify run-time options for your application using one of the following methods:

- As default options established in CEEDOPT during the installation of LE/VSE (see page 35 for more information about CEEDOPT)
- As application defaults established in CEEUOPT (see page 36 and "CEEXOPT Invocation Syntax" on page 40 for details)
- In the user exit (see "CEEBXITA Assembler User Exit Interface" on page 323 for details on how to do this)
- In C applications, as options specified using `#pragma runopts` (see page 36 for the proper syntax)
- In PL/I applications, as default options established in CEEUOPT using the PLIXOPT string (see page 37 for the proper syntax)

Condition Handling with DL/I

The DL/I environment is sensitive to errors or conditions. A failing DL/I transaction or application can potentially contaminate a DL/I database. For this reason, it is essential that DL/I knows about the failure of a transaction or application that has been updating a database so that it can perform all necessary cleanup activities. These include database rollback (the backing out of any updates made by a failing online transaction), writing back any pending I/O buffers to the physical database, and completing the journal file (if logging has been active).

Under CICS, DL/I database recovery is managed by the CICS-DL/I interface. For more information about condition handling under CICS, see "Ensuring Transaction Rollback under CICS" on page 307.

In the batch environment, DL/I database recovery is managed by the LE/VSE-DL/I interface. When you run your batch application with the TRAP(ON) run-time option, LE/VSE and DL/I DOS/VS keep track of calls to and returns from DL/I. If a program interrupt or abend occurs when your application is running, the LE/VSE condition manager is informed whether the problem occurred in your application or in DL/I. If the program interrupt or abend occurs in DL/I, the LE/VSE condition handler percolates the condition back to DL/I to allow DL/I to do the required cleanup.

If a program interrupt or abend occurs in the application outside of DL/I, or a software condition of severity 2 or greater is raised outside of DL/I, the condition manager takes normal condition handling actions as described in Chapter 11, “LE/VSE Condition Handling Introduction,” on page 103. If the condition manager gets control then you **must** do one of the following:

- Resolve the error completely so that the application can continue.
- Make sure that the application terminates abnormally by using the ABTERMENC(ABEND) run-time option to transform all abnormal terminations into operating system abends in order to allow the DL/I exit to do the required cleanup.
- Make sure that the application terminates abnormally by coding and providing a modified run-time assembler user exit (CEEBXITA) that transforms all abnormal terminations into operating system abends in order to allow the DL/I exit to do the required cleanup. The assembler user exit you provide should check the return code and reason code or the CEEAUE_ABTERM bit, and request an abend by setting the CEEAUE_ABND flag to ON, if appropriate. See “CEEBXITA Assembler User Exit Interface” on page 323 for more details about CEEBXITA user exit.

Note: If a program interrupt or abend occurs, regardless of whether it occurs within DL/I, or in the application outside DL/I, DL/I cleanup is only performed when DL/I exit processing has been enabled through the appropriate UPSI byte setting using the UPSI job control statement.

Part 5. Specialized Programming Tasks

Chapter 25. Using Run-Time User Exits 319
Understanding the Basics 319
User Exits Supported under LE/VSE 319
Using the Assembler User Exit CEEBXITA. . . 320
Using the HLL Initialization Exit CEEBINT . . 320
Using Sample Assembler User Exits 320
When User Exits Are Invoked 321
CEEBXITA Assembler User Exit Interface . . 323
CEEBINT High-Level Language User Exit Interface 332

Chapter 26. Assembler Considerations 335
Understanding the Basics 335
Compatibility Considerations 335
Register Conventions. 335
Considerations for Coding or Running Assembler Routines 336
Condition Handling 336
Access to the Inbound Parameter String . . . 336
Overlay Programs. 337
CEESTART, CEEMAIN, and CEEFMAIN . . . 337
LE/VSE Library Routine Retention 337
Using Library Routine Retention 338
Library Routine Retention and Preinitialization 338
CEELRR Macro— Initialize/Terminate LE/VSE Library Routine Retention 339
Assembler Macros. 341
CEEENTRY Macro— Generate an LE/VSE-Conforming Prolog 341
CEETERM Macro— Terminate an LE/VSE-Conforming Routine 343
CEECAA Macro— Generate a CAA Mapping 344
CEECIB Macro— Generate a CIB Mapping . . 345
CEEDSA Macro— Generate a DSA Mapping 345
CEEPPA Macro— Generate a PPA 345
CEELOAD Macro— Dynamically Load a Routine 348
Usage Notes. 350
CEEFETCH Macro— Dynamically Load a Routine that Can Be Later Deleted 350
Usage Notes. 352
CEERELES Macro— Dynamically Delete a Routine 353
Usage Notes. 354
Example of Assembler Main Routine 355
Example of an Assembler Main Calling an Assembler Subroutine 356
Invoking Callable Services from Assembler Routines 359
System Services Available to Assembler Routines 359

Chapter 27. Using Preinitialization Services . . 363
Understanding the Basics 363
Compatibility 364
COBOL 364
Using Preinitialization 364
Using the PIPI Table 364
C Considerations 364
COBOL Considerations 364
PL/I Considerations 365
Macros that Generate the PIPI Table. . . . 365
Reentrancy Considerations 366
User Exit Invocation 366
Stop Semantics 367
Specifying Run-Time Options and Program Arguments 367
CEEPIPI Interface 368
Initialization. 369
CEEPIPI(init_main)—Initialize for Main Routines 369
CEEPIPI(init_sub)—Initialize for Subroutines 370
CEEPIPI(init_sub_dp)—Initialize for Subroutines (Multiple Environment). . . . 371
Application Invocation 373
CEEPIPI(call_main)—Invocation for Main Routine 373
CEEPIPI(call_sub)—Invocation for Subroutines 374
CEEPIPI(call_sub_addr)—Invocation for Subroutines by Address 376
CEEPIPI(start_seq)—Start a Sequence of Calls 377
CEEPIPI(end_seq)—End a Sequence of Calls 378
Termination 379
CEEPIPI(term)—Terminate Environment . . 379
Adding an Entry to the PIPI Table 380
CEEPIPI(add_entry)—Add an Entry to the PIPI Table 380
Deleting an Entry from the PIPI Table 381
CEEPIPI(delete_entry)—Delete an Entry from the PIPI Table 381
Service Routines 382
An Example Program Invocation of CEEPIPI . . . 387
HLLPIPI Examples 390

Chapter 28. Using Nested Enclaves 393
Understanding the Basics 393
COBOL Considerations 393
Determining the Behavior of Child Enclaves . . . 393
Creating Child Enclaves Using EXEC CICS LINK or EXEC CICS XCTL 394
How Run-Time Options Affect Child Enclaves 394
How Conditions Arising in Child Enclaves Are Handled 394
Creating Child Enclaves Using the C system() Function 395
How Conditions Arising in Child Enclaves Are Handled 395
Other Nested Enclave Considerations 396
What the Enclave Returns from CEE5PRM . . 396
Finding the Return and Reason Code from the Enclave 397

© Copyright IBM Corp. 1991, 2005

Assembler User Exit 397
MSGFILE Considerations 397
AMODE Considerations 397

The chapters in this section describe advanced or specialized tasks that you can perform in LE/VSE.

Chapter 25. Using Run-Time User Exits

LE/VSE provides user exits that you can use for functions at your installation. You can use the assembler user exit (CEEBXITA) or the HLL user exit (CEEBINT). This chapter provides information about using these run-time user exits.

Understanding the Basics

User exits are invoked under LE/VSE to perform enclave initialization functions and both normal and abnormal termination functions. User exits offer you a chance to perform certain functions at a point where you would not otherwise have a chance to do so. In an assembler initialization user exit, for example, you can specify a list of run-time options that establish characteristics of the environment. This is done prior to the actual execution of any of your application code.

In most cases, you do not need to modify any user exit in order to run your application. Instead, you can accept the IBM-supplied default versions of the exits, or the defaults as defined by your installation. To do so, run your application in the normal manner and the default versions of the exits are invoked. You might also want to read the sections "User Exits Supported under LE/VSE" and "When User Exits Are Invoked" on page 321, which provide an overview of the user exits and describe when they are invoked.

If you plan to modify either of the user exits to perform some specific function, you must link the modified exit to your application before running. In addition, the sections "Using the Assembler User Exit CEEBXITA" on page 320 and "CEEBINT High-Level Language User Exit Interface" on page 332 describe the respective user exit interfaces to which you must adhere in order to change an assembler or HLL user exit.

User Exits Supported under LE/VSE

LE/VSE provides two user exit routines, one written in assembler (CEEBXITA) and the other in an LE/VSE-conforming HLL (CEEBINT). If LE/VSE is installed in the default sublibraries, you can find the IBM-supplied default exits in the PRD2.SCEEBASE sublibrary.

The user exits supported by LE/VSE are shown in Table 47.

Table 47. User Exits Supported under LE/VSE

Name	Type of User Exit	When Invoked
CEEBXITA	Assembler user exit	Enclave initialization Enclave termination Process termination
CEEBINT	HLL user exit. CEEBINT can be written in C, PL/I, or LE/VSE-conforming assembler.	Enclave initialization

When CEEBXITA or CEEBINT is linked with the LE/VSE initialization/termination library routines during installation, it functions as an installation-wide user exit. When CEEBXITA or CEEBINT is linked in your phase, it functions as an

© Copyright IBM Corp. 1991, 2005

application-specific user exit. The application-specific exit is used only when you run that application. The installation-wide assembler user exit is not executed.

To use an application-specific user exit, you must explicitly include it at link-edit time in the application phase using a linkage editor INCLUDE control statement (see "Using the INCLUDE Statement" on page 26 for more information). Any time that the application-specific exit is modified, it must be relinked with the application.

The assembler user exit interface is described in "CEEBXITA Assembler User Exit Interface" on page 323. The HLL user exit interface is described in "CEEBINT High-Level Language User Exit Interface" on page 332.

Using the Assembler User Exit CEEBXITA

CEEBXITA tailors the characteristics of the enclave prior to its establishment. It must be written in assembler language because an HLL environment is not yet established when the exit is invoked. CEEBXITA is driven for enclave initialization and enclave termination regardless of whether the enclave is the first enclave in the process or a nested enclave. CEEBXITA can differentiate easily between first and nested enclaves. For more information about nested enclaves, see Chapter 28, "Using Nested Enclaves," on page 393.

CEEBXITA is invoked very early during the initialization process, before enclave initialization is complete. The enclave initialization code recognizes run-time options specified by CEEBXITA.

Using the HLL Initialization Exit CEEBINT

CEEBINT is invoked just before the invocation of the application code. In LE/VSE, this exit can be written in C, PL/I, or in LE/VSE-conforming assembler. CEEBINT **cannot** be written in COBOL, even though COBOL applications can use this HLL user exit. When CEEBINT is invoked, the run-time environment is fully operational and all LE/VSE-conforming HLLs are supported.

Using Sample Assembler User Exits

You can use the sample assembler user exit programs distributed with LE/VSE to modify the code for the requirements of your application. Choose a sample program appropriate for your application. The following assembler user exit programs are delivered with LE/VSE:

Table 48. Sample Assembler User Exits for LE/VSE

Example User Exit Member Name	Operating Environment	Language (if Language-Specific)
CEEBXITA.A	VSE (default)	
CEECXITA.A	CICS (default)	
CEEBX05A.A	VSE	VS COBOL II compatibility

Note:

1. If LE/VSE is installed at your site without modification, then CEEBXITA and CEECXITA are the defaults on your system for VSE and CICS, respectively.

If LE/VSE is installed in the default sublibraries, you can find the source code for CEEBXITA, CEECXITA, and CEEBX05A in the PRD2.SCEEBASE sublibrary.

The assembler user exit CEEBXITA performs functions for enclave initialization, normal and abnormal enclave termination, and process termination. CEEBXITA must be written in assembler language, because an HLL environment might not be established when the exit is invoked.

You can set up user exits for tasks such as:

Installation accounting and charge back
Installation audit controls
Programming standard enforcement
Common application run-time support

When User Exits Are Invoked

Figure 115 shows the timing of the invocations of the user exits at initialization and termination processing.

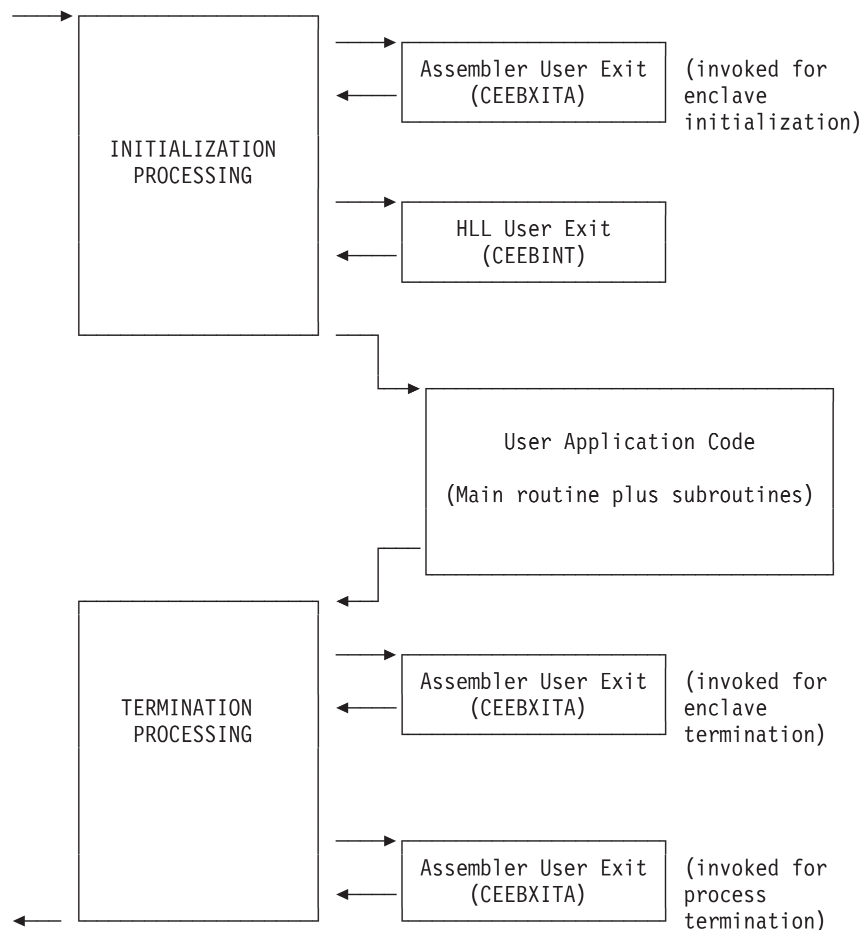


Figure 115. Location of User Exits

In Figure 115, run-time user exits are invoked in the following sequence:

1. Assembler user exit is invoked for enclave initialization
2. Environment is established
3. HLL user exit is invoked
4. Main routine is invoked
5. Main routine returns control to caller
6. Assembler user exit is invoked for termination of the enclave

CEEBXITA is invoked for enclave termination processing after all application code in the enclave has completed, but prior to any enclave termination activity.

7. Environment is terminated
8. Assembler user exit is invoked for termination of the process

   CEEBXITA is invoked again when the LE/VSE process terminates.

LE/VSE provides the CEEBXITA assembler user exit for termination but does not provide a corresponding HLL termination user exit.

CEEBXITA behaves differently, depending upon when it is invoked, as described in the following sections.

CEEBXITA Behavior During Enclave Initialization: The CEEBXITA assembler user exit is invoked before enclave initialization is performed. You can use CEEBXITA to help establish your application run time environment. For example, in the assembler user exit you can specify the stack and heap run-time options. You can also use the user exit to interrogate program parameters supplied in the JCL and change them if you want. In addition, you can specify run-time options in the user exit by using the CEEAUE_OPTION field of the assembler interface.

CEEBXITA Behavior During Enclave Termination: The CEEBXITA assembler exit is invoked after the user code for the enclave has completed, but prior to the occurrence of any enclave termination activity. In other words, the assembler user exit for termination is invoked when the environment is still active. For example, CEEBXITA is invoked before the storage report is produced (if you requested one), files are closed, and the debug tool is invoked for enclave termination.

The assembler user exits permit you to request an abend. You can also request a dump to assist in problem diagnosis. Because termination activities have not yet begun when the user exit is invoked, the majority of storage has not been modified when the dump is produced.

You can request the abend and dump in the assembler user exit for all enclave-terminating events including:

- The situation that occurs in PL/I when the ON condition (including ERROR or FINISH) is raised and one of the following conditions is true:
  - The program does not have an appropriate ON-unit.
  - The ON-unit does not terminate with a GOTO.
  - The GOTO is not allowed.

  This rule applies only to the conditions that cause termination of the program.
- Return from the main routine
- A debug tool QUIT command
- An HLL stop statement such as:
  - C `exit()`
  - COBOL STOP RUN
  - PL/I STOP or EXIT
- An unhandled condition of severity 2 or above

CEEBXITA Behavior During Process Termination: The CEEBXITA assembler exit is invoked after:

All enclaves have terminated
The enclave resources have been relinquished
Any LE/VSE-managed files have been closed

The debug tool has terminated

At this time you can free allocated files and request an abend.

During termination, CEEBXITA can interrogate the LE/VSE reason and return codes and, if necessary, request an abend with or without a dump. This can be done at either enclave or process termination.

Specifying Abnormal Conditions to Be Exempted from Condition Handling: The assembler user exit, when invoked for initialization in the batch environment, can return a list of VSE cancel codes, program-interruption codes, and user abend codes (contained in the CEEAUE_CODES field of the assembler user exit interface—see "CEEBXITA Assembler User Exit Interface") that are to be exempted from LE/VSE condition handling.

When an abend or program interrupt occurs in your application, and TRAP(ON,MAX) is in effect, and the VSE cancel code, program-interruption code, or user abend code is in the CEEAUE_CODES list, LE/VSE produces an abnormal termination message and issues an abend to terminate the enclave. Normal LE/VSE condition handling is never invoked to handle these conditions. This feature is useful when you do not want LE/VSE condition handling to intervene for certain abends, and when you want to produce a system dump.

When TRAP(ON,MIN) is specified and there is a program interrupt, the user exit for termination is not driven.

Actions Taken for Errors That Occur within the Assembler User Exit: If any errors occur during the enclave initialization user exit, the standard system action occurs because LE/VSE condition handling has not yet been established.

Any errors occurring during the enclave termination user exit lead to abnormal termination (through an abend) of the LE/VSE environment.

If there is a program check during the enclave termination user exit and TRAP(ON,MAX) is in effect, the application ends abnormally with message CEE3322C and user abend code 4094 and reason code 44. If there is a program check during the enclave termination user exit and TRAP(ON,MIN) has been specified, the application ends abnormally without additional error checking support. LE/VSE performs no condition handling; error handling is performed by the operating system.

If there is a program check during the process termination user exit, the application ends abnormally without additional error checking support, regardless of the setting of the TRAP run-time option. LE/VSE performs no condition handling; error handling is performed by the operating system.

CEEBXITA Assembler User Exit Interface

You can modify CEEBXITA to perform any function you need, but the exit must have the following attributes after you modify it at installation:

- The user-supplied exit must be named CEEBXITA.
- The exit must be reentrant.
- The exit must be capable of executing in AMODE(ANY) and RMODE(ANY).
- The exit must be relinked with LE/VSE initialization/termination routines after modification.

If a user exit is modified, you are responsible for conforming to the interface shown in Figure 116. Note that this user exit **must** be written in assembler.

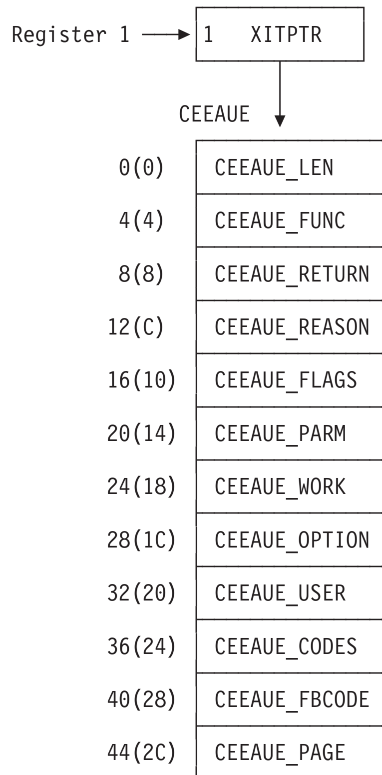


Figure 116. Interface for CEEBXITA Assembler User Exit

When the user exit is called, register 1 points to a word that contains the address of the CEEAUE control block. The high-order bit is on.

The CEEAUE control block contains the following fullwords:

CEEAUE_LEN (input parameter)
: A fullword integer that specifies the total length of this control block. For LE/VSE, the length is 48 bytes.

CEEAUE_FUNC (input parameter)
: A fullword integer that specifies the function code. LE/VSE supports the following function codes:

- **1** Initialization of the first enclave within a process.
- **2** Termination of the first enclave within a process.
- **3** Nested enclave initialization.
- **4** Nested enclave termination.
- **5** Process termination.

The user exit should ignore function codes other than those numbered from 1 through 5.

CEEAUE_RETURN (input/output parameter)
: A fullword integer that specifies the return or abend code. CEEAUE_RETURN has different meanings, depending on whether it is an input parameter or an output parameter:

- As an input parameter, CEEAUE_RETURN is the enclave return code.
- As an output parameter, CEEAUE_RETURN has different meanings, depending on the flag CEEAUE_ABND (see below):
  - If the flag CEEAUE_ABND is off, CEEAUE_RETURN is interpreted as the LE/VSE return code placed in register 15.
  - If the flag CEEAUE_ABND is on, CEEAUE_RETURN is interpreted as an abend code used when an abend is issued. (In batch, run-time message CEE3322C is produced and an operating system request is issued to terminate the enclave; in CICS, an EXEC CICS ABEND is issued.)

CEEAUE_REASON (input/output parameter)
A fullword integer that specifies the reason code for CEEAUE_RETURN. CEEAUE_REASON has different meanings, depending on whether it is an input parameter or an output parameter:
- As an input parameter, CEEAUE_REASON is the LE/VSE return code modifier.
- As an output parameter, CEEAUE_REASON has different meanings, depending on the flag CEEAUE_ABND (see below):
  - If the flag CEEAUE_ABND is off, CEEAUE_REASON is interpreted as the LE/VSE return code modifier placed in register 0.
  - If the flag CEEAUE_ABND is on, CEEAUE_REASON is interpreted as an abend reason code used when an abend is issued. (CEEAUE_REASON is used in the batch abnormal-termination run-time message CEE3322C, but is ignored in the CICS environments when an EXEC CICS ABEND is issued.)

CEEAUE_FLAGS
Contains four 1-byte flags. CEEBXITA uses only the first byte but reserves the remaining flags. All unspecified bits and bytes must be 0. The layout of these flags is shown in Figure 117:

```
Byte 0
        x... .... - CEEAUE_ABTERM
        0... .... - Normal termination
        1... .... - Abnormal termination
        .x.. .... - CEEAUE_ABND
        .0.. .... - Terminate with CEEAUE_RETURN
        .1.. .... - ABEND with CEEAUE_RETURN and
                      CEEAUE_REASON given
        ..x. .... - CEEAUE_DUMP
        ..0. .... - If CEEAUE_ABND=1 ABEND with no dump
        ..1. .... - If CEEAUE_ABND=1 ABEND with a dump
        ...0 0000 - Reserved bits (must be zero)
Byte 1
        00 - Reserved for future use
Byte 2
        00 - Reserved for future use
Byte 3
        00 - Reserved for future use
```

Figure 117. CEEAUE_FLAGS Format

Byte 0 (CEEAUE_FLAG1) has the following meaning:

CEEAUE_ABTERM (input parameter)

OFF Indicates that the enclave is terminating normally (severity 0 or 1 condition).

ON Indicates that the enclave is terminating with an LE/VSE return code modifier of 2 or greater. This could, for example, indicate that a severity 2 or greater condition was raised but not handled.

CEEAUE_ABND (input/output parameter)

OFF Indicates that the enclave should terminate without an abend being issued. Thus, CEEAUE_RETURN and CEEAUE_REASON are placed into register 15 and register 0 respectively and returned to the enclave creator.

ON Indicates that the enclave terminates with an abend. Thus, CEEAUE_RETURN and CEEAUE_REASON are used by LE/VSE in the invocation of the abend. When running in the batch environment, run-time message CEE3322C is produced and an operating system request is issued to terminate the enclave. When running under CICS, an EXEC CICS ABEND command is issued using the abend code contained in CEEAUE_RETURN. CEEAUE_REASON is ignored under CICS.

The TRAP run-time option does not affect the setting of CEEAUE_ABND.

When the ABTERMENC(ABEND) run-time option is specified, the enclave always terminates with an abend when there is an unhandled condition of severity 2 or greater, regardless of the setting of the CEEAUE_ABND flag. However, if you want a system dump to be produced when the enclave terminates with an abend, you must set CEEAUE_ABND and CEEAUE_DUMP to ON. See “Termination Behavior for Unhandled Conditions” on page 70 for a detailed explanation of how the CEEAUE_ABND parameter can affect the behavior of the ABTERMENC run-time option.

CEEAUE_DUMP (output parameter)

OFF Indicates that when you request an abend, by setting CEEAUE_ABND to ON, an abend is issued without requesting a dump.

ON Indicates that when you request an abend by setting CEEAUE_ABND to ON, an abend requesting a dump is issued. You must also specify the VSE DUMP option if you want a system dump to be produced when you request an abend.

CEEAUE_PARM (input/output parameter)
A fullword pointer to the parameter address list of the application program.

As an input parameter, CEEAUE_PARM contains the register 1 value passed to the main routine. The exit can modify this value, and the value is then passed to the main routine. If run-time options are present in the PARM parameter of the JCL EXEC statement, they are stripped off before the exit is called.

If the parameter inbound to the main routine is a character string, CEEAUE_PARM contains the address of a fullword address that points to a halfword prefixed string. The halfword prefix contains the length of the string. If no program arguments are specified in the PARM parameter of the JCL EXEC statement, the halfword prefix contains zero.

Note: It is important that before using this pointer your program checks:
- if CEEAUE_PARM has a non-zero value.
- the length and content of invocation string.

If this string is altered by the user exit, the string must not be extended in place. Before the string is extended, it must be copied to an area of user storage large enough for the extended string.

CEEAUE_WORK (input parameter)
A fullword pointer to a 256-byte work area that the exit can use. On entry it contains binary zeros and is doubleword-aligned.

This area does not persist across exits.

CEEAUE_OPTION (output parameter)
Upon return, CEEAUE_OPTION contains a fullword pointer to the address of a halfword-length prefixed character string that contains run-time options. These options are honored only during the initialization of the first enclave. When invoked for enclave termination, CEEAUE_OPTION is ignored.

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

Under CICS, the STACK run-time option cannot be modified with the assembler user exit.

CEEAUE_USER (input/output parameter)
A fullword whose value is maintained without alteration and passed to every user exit. Upon entry to the enclave initialization user exit, it is zero. Thereafter, the value of the user word is not altered by LE/VSE or any member libraries. The user exit might change the value of CEEAUE_USER, and LE/VSE maintains that value. This allows the initialization user exit to acquire and initialize a work area, save its address in CEEAUE_USER, and pass the work area address to subsequent user exits. The work area might be freed by the termination user exit.

CEEAUE_CODES (output parameter)
During the initialization exit, CEEAUE_CODES contains the fullword address of a table of VSE cancel codes, program-interruption codes, and user-abend codes that the LE/VSE condition handler exempts from normal condition handling. Therefore, the application is not given the opportunity to field the abend. The table consists of:

- A fullword count of the number of cancel codes, program-interruption codes, and abend codes that are to be exempted from LE/VSE condition handling, and passed to the operating system.
- A fullword for each of the particular cancel codes, program-interruption codes, or abend codes that are to be exempted from LE/VSE condition handling, and passed to the operating system.
  - User abend codes are specified as F'uuuu'. For example, if you want user abend 7777 to be exempted from LE/VSE condition handling, code F'7777'.
  - VSE cancel codes are specified as X'000000cc'. Avoid specifying the value X'00000020', which indicates a program check has occurred. If you specify the value X'00000020', LE/VSE ignores it, and normal LE/VSE condition handling semantics take effect. If you want to exempt specific program checks from LE/VSE condition handling, specify the program-interruption codes.
  - Program-interruption codes are specified as X'800000ii'. For example, if you want an operation exception to be exempted from LE/VSE condition handling, code X'80000001'.

This function is not enabled under CICS.

CEEAUE_FBCODE (input parameter)
Contains a fullword address of the condition token with which the enclave terminated. If the enclave terminates normally (that is, not due to a condition), the condition token is zero.

CEEAUE_PAGE (input parameter)
This parameter indicates whether PL/I BASED variables that are allocated storage outside of AREAs are allocated on a 4K-page boundary. You can specify in the field the minimum number of bytes of storage that must be allocated. Your allocation request must be an exact multiple of 4K.

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

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

CEEAUE_PAGE is honored only during enclave initialization, that is, when CEEAUE_FUNC is 1 or 3.

Parameter Values in the Assembler User Exit: The parameters described in "CEEBXITA Assembler User Exit Interface" on page 323 contain different values depending on how the user exit is used. Table 49 on page 329 and Table 50 on page 331 describe the possible values for the parameters based on how the assembler user exit is invoked.

Table 49. Parameter Values in the Assembler User Exit (Part 1). The assembler user exit contains these parameter values depending on when it is invoked.

When Invoked	CEEAUE_ LEN	CEEAUE_RETURN	CEEAUE_ REASON (See Note 1)	CEEAUE_ FLAGS	CEEAUE_PARM
First Enclave within Process Initialization — Entry CEEAUE_FUNC = 1	48	0	0	0	The address of a fullword that points to a string of user parameters prefixed by a halfword length. If no parameters are present, the halfword length contains zero. You can alter the string in a user exit. Upon return, the CEEAUE_PARM is processed and merged as the invocation string.
First Enclave within Process Initialization — Return		0, or abend code if CEEAUE_ABND = 1	0, or reason code for CEEAUE_RETURN if CEEAUE_ABND = 1	See Note 2 on page 330.	The address of a fullword that points to an optionally altered string of user parameters prefixed by a halfword length. If no parameters are present, the halfword length contains zero. Upon return, the CEEAUE_PARM is processed and merged as the invocation string.
First Enclave within Process Termination — Entry CEEAUE_FUNC = 2	48	Return code issued by application that is terminating.	Reason code that accompanies CEEAUE_RETURN.	See Note 3 on page 330.	
First Enclave within Process Termination — Return		If CEEAUE_ABND = 0, the return code placed into register 15 when the enclave terminates. If CEEAUE_ABND = 1, the abend code.	If CEEAUE_ABND = 0, the enclave reason code. If CEEAUE_ABND = 1, the abend reason code.	See Note 2 on page 330.	
Nested Enclave Initialization — Entry CEEAUE_FUNC = 3	48	0	0	0	The address of a fullword that points to a string of user parameters prefixed by a halfword length. If no parameters are present, the halfword length contains zero. You can alter the string in a user exit. Upon return, the CEEAUE_PARM is processed and merged as the invocation string.
Nested Enclave Initialization — Return		0, or if CEEAUE_ABND = 1, the abend code.	0, or if CEEAUE_ABND = 1, reason code for CEEAUE_RETURN.	See Note 2 on page 330.	The address of a fullword that points to an optionally altered string of user parameters prefixed by a halfword length. If no parameters are present, the halfword length contains zero. Upon return, the CEEAUE_PARM is processed and merged as the invocation string.
Nested Enclave Termination — Entry CEEAUE_FUNC = 4	48	Return code issued by enclave that is terminating.	Reason code accompanying CEEAUE_RETURN.	See Note 3 on page 330.	

Table 49. Parameter Values in the Assembler User Exit (Part 1) (continued). The assembler user exit contains these parameter values depending on when it is invoked.

When Invoked	CEEAUE_ LEN	CEEAUE_RETURN	CEEAUE_ REASON (See Note 1)	CEEAUE_ FLAGS	CEEAUE_PARM
Nested Enclave Termination — Return		If CEEAUE_ABND = 0, the return code from the enclave. If CEEAUE_ABND = 1, the abend code.	If CEEAUE_ABND = 0, the enclave reason code. If CEEAUE_ABND = 1, the enclave reason code.	See Note 2.	
Process Termination — Entry Function Code = 5	48	Return code presented to the invoking system in register 15 that reflects the value returned from the "first enclave within process termination".	Reason code accompanying CEEAUE_RETURN that is presented to the invoking system in register 0 and reflects the value returned from the "first enclave within process termination".	See Note 4.	
Process Termination — Return		If CEEAUE_ABND = 0, return code from the process. If CEEAUE_ABND = 1, the abend code.	If CEEAUE_ABND = 0, the reason code for CEEAUE_RETURN from the process. If CEEAUE_ABND = 1, reason code for the CEEAUE_RETURN abend reason code.	See Note 2.	

Notes:

1. CEEAUE_REASON is ignored under CICS when CEEAUE_ABND = 1.
2. **CEEAUE_FLAGS:**
   CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing
   CEEAUE_DUMP = 1 if the abend should request a dump
3. **CEEAUE_FLAGS:**
   CEEAUE_ABTERM = 1 if the application is terminating with an LE/VSE return code modifier of 2 or greater, or 0 otherwise
   CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing
   CEEAUE_DUMP = 0
4. **CEEAUE_FLAGS:**
   CEEAUE_ABTERM = 1 if the last enclave is terminating abnormally (that is, an LE/VSE return code modifier is 2 or greater). This reflects the value returned from the "first enclave within process termination".
   CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing "first enclave within process termination" (function code 2).
   CEEAUE_DUMP = 0

Table 50. Parameter Values in the Assembler User Exit (Part 2). The assembler user exit contains these parameter values depending on when it is invoked.

When Invoked	CEEAUE_WORK	CEEAUE_OPTION	CEEAUE _USER	CEEAUE _ CODES	CEEAUE _FBCODE	CEEAUE_PAGE
First Enclave within Process Initialization — Entry CEEAUE_FUNC = 1	Address of a 256-byte work area of binary zeros.		0		0	Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).
First Enclave within Process Initialization — Return		Pointer to address of a halfword prefixed character string containing run-time options, or 0.	Value of CEEAUE_USER for all subsequent exits.	Pointer to the abend codes table, or 0.		User specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).
First Enclave within Process Termination — Entry CEEAUE_FUNC = 2	Address of a 256-byte area of binary zeros.		Return value from previous exit.		Feedback code causing termination.	
First Enclave within Process Termination — Return			The value of CEEAUE_USER for all subsequent exits.			
Nested Enclave Initialization — Entry CEEAUE_FUNC = 3	Address of a 256-byte work area of binary zeros.		Return value from previous exit.		0	Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).
Nested Enclave Initialization — Return		Pointer to fullword address that points to a halfword prefixed length string containing run-time options, or 0.	The value of CEEAUE_USER for all subsequent exits.	Pointer to abend codes table, or 0.		User specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).
Nested Enclave Termination — Entry CEEAUE_FUNC = 4	Address of a 256-byte work area of binary zeros.		Return value from previous exit.		Feedback code causing termination.	
Nested Enclave Termination — Return			Value of CEEAUE_USER for all subsequent exits.			
Process Termination — Entry CEEAUE_FUNC = 5	Address of a 256-byte work area of binary zeros.		Return value from previous exit.		Feedback code causing termination.	
Process Termination — Return			Value of CEEAUE_USER for all subsequent exits.			

CEEBINT High-Level Language User Exit Interface

LE/VSE provides CEEBINT for enclave initialization. You can code CEEBINT in C, PL/I, or LE/VSE-conforming assembler. COBOL programs can use CEEBINT, but CEEBINT cannot be written in COBOL or be used to call COBOL programs.

You can modify CEEBINT to perform any function desired, although the exit must have the following attributes after you modify it:

- The user exit must not be a main-designated routine. That is, it must not be a C main function, and OPTIONS(MAIN) must not be specified for PL/I applications.
- CEEBINT must be linked with compiled code. If you do not provide an initialization user exit, an IBM-supplied default, which simply returns control to your application, is linked with the compiled code.
- The exit cannot be written in COBOL.
- The exit should be coded so that it returns for all unknown function codes.
- C constructs such as the `exit()`, `abort()`, `raise(SIGTERM)`, and `raise(SIGABRT)` functions terminate the enclave.
- A PL/I EXIT or STOP statement terminates the enclave.
- Use the callable service IBMHKS to turn hooks on and off. For more information about IBMHKS, see *IBM PL/I for VSE/ESA Programming Guide*
- C functions such as `exit()`, `abort()`, `raise(SIGTERM)`, and `raise(SIGABRT)` terminate the entire application as well as the user exit.

CEEBINT is invoked after the enclave has been established, after the debug tool initial command string has been processed, and prior to the invocation of compiled code. When invoked, it is passed a parameter list. The parameters are all fullwords and are defined as:

Number of arguments in parameter list (input)
A fullword binary integer.
- On entry: Contains 7.
- On exit: Not applicable.

Return code (output)
A fullword binary integer.
- On entry: 0.
- On exit: Able to be set by the exit, but not interrogated by LE/VSE.

Reason code (output)
A fullword binary integer.
- On entry: 0.
- On exit: Able to be set by the exit, but not interrogated by LE/VSE.

Function code (input)
A fullword binary integer.
- On entry: 1, indicating the exit is being driven for initialization.
- On exit: Not applicable.

Address of the main program entry point (input)
A fullword binary address.
- On entry: The address of the routine that gains control first.
- On exit: Not applicable.

User word (input/output)
A fullword binary integer.

- On entry: Value of the user word (CEEAUE_USER) as set by the assembler user exit. See page 327 for a description of the CEEAUE_USER field.
- On exit: The value set by the user exit, maintained by LE/VSE and passed to subsequent user exits.

Exit List Address (output)
The address of the exit list control block, Exit_list.

- On entry: 0.
- On exit: 0, unless you establish a hook exit, in which case you would set this pointer and fill in relevant control blocks. The control blocks for Exit_list and Hook_exit are shown in the following figure.

As supplied, CEEBINT has only one exit defined that you can establish—the hook exit described by the Hook_exit control block. This exit gains control when hooks generated by the PL/I compile-time TEST option are executed. You can establish this exit by setting appropriate pointers (A_Exits to Exit_list to Hook_exit).

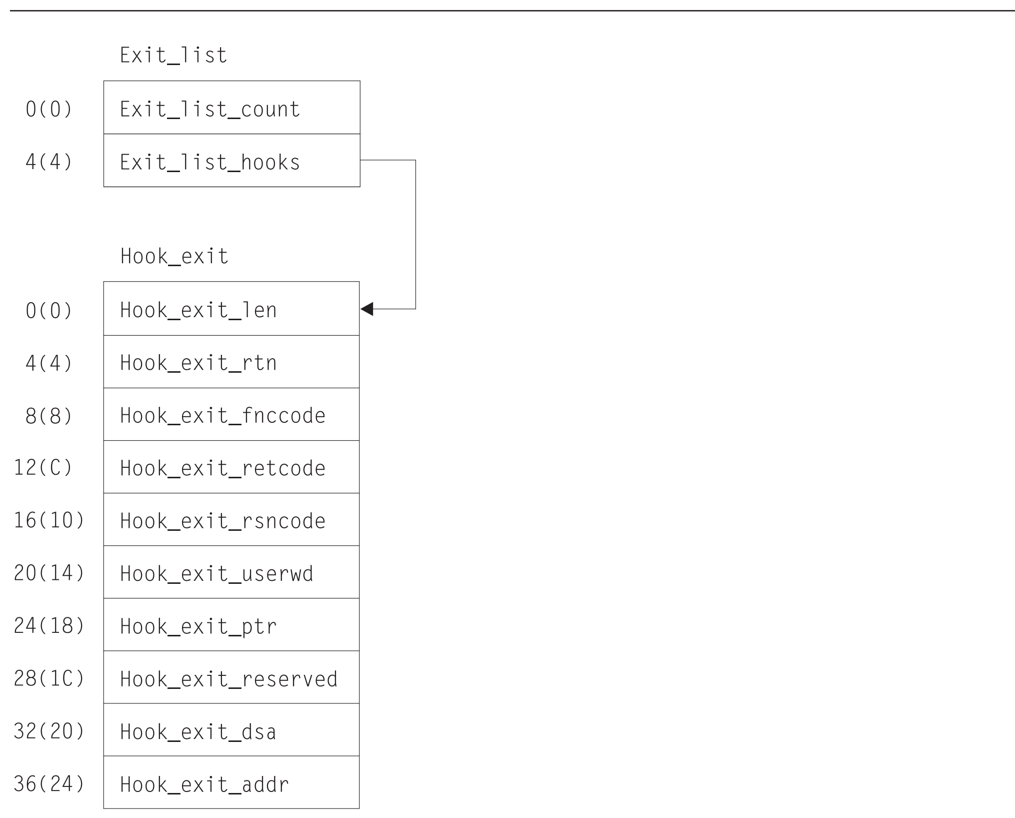


Figure 118. Exit_list and Hook_exit Control Blocks

The control block Exit_list exit contains the following fields:

Exit_list_count
The count of exit list hooks. It must be 1.

Exit_list_hooks
The address of the Hook_exit control block.

The control block for the hook exit must contain the following fields:

Hook_exit_len
The length of the control block.

Hook_exit_rtn
The address of a routine you want invoked for the exit. When the routine is invoked, it is passed the address of this control block. Since this routine is invoked only if the address you specify is nonzero, you can turn the exit on and off.

Hook_exit_fnccode
The function code with which the exit is invoked. This is always 1.

Hook_exit_retcode
The return code set by the exit. You must ensure it conforms to the following specifications:

0 Requests that the debug tool be invoked next

4 Requests that the program resume immediately

16 Requests that the program be terminated

Hook_exit_rsncode
The reason code set by the exit. This is always zero.

Hook_exit_userwd
Reserved.

Hook_exit_ptr
An exit-specific user word.

Hook_exit_reserved
Reserved.

Hook_exit_dsa
The contents of register 13 when the hook was executed.

Hook_exit_addr
The address of the hook instruction executed.

Chapter 26. Assembler Considerations

You can run applications written in assembler language in LE/VSE. Applications written in LE/VSE-conforming HLLs can also call or be called by assembler language applications.

This chapter discusses considerations for assembler applications and introduces library routine retention, a function that can provide performance improvement for applications running in the batch environment.

Understanding the Basics

Whether you plan to execute a single-language assembler application or a multiple-language application containing assembler code, there are a number of restrictions you must follow under LE/VSE.

For example, to communicate with LE/VSE and other applications running in the common run-time environment, your assembler application must preserve the use of certain registers and storage areas in a consistent way. Calling conventions for assembler programs must follow the standard S/370 linkage conventions. In addition, your assembler program is restricted from using some operating system services. These conventions and restrictions are described in this chapter.

Compatibility Considerations

If you are coding a new assembler routine that you want to conform to the LE/VSE interface or if your assembler routine calls LE/VSE services, you must use the macros provided by LE/VSE. For a list of these macros, see "Assembler Macros" on page 341. Throughout this book, *LE/VSE-conforming assembler routine* refers to an assembler routine coded using the CEEENTRY and associated macros.

Assembler routines that rely on control blocks that were valid under previous versions of C, COBOL, and PL/I (for example, routines that check flags or switches in these control blocks) might be invalid under LE/VSE. These control blocks might have changed. For more information, see one of the migration guides listed in "Where to Find More Information" on page xxi.

Any assembler routine used within the scope of an LE/VSE application must use standard save area conventions, and the save area address in R13 must be a valid 31-bit address.

LE/VSE-conforming assembler main routines are not supported under CICS.

Register Conventions

To communicate properly with assembler routines, you must observe certain register conventions on entry into the assembler routine (while it runs), and on exit from the assembler routine. These conventions are honored when you use the macros listed in "Assembler Macros" on page 341 to write your assembler application or if you call any LE/VSE service.

On entry into the assembler main routine, registers must contain the following values because they are passed without change to the CEEENTRY macro:

© Copyright IBM Corp. 1991, 2005

R0 Undefined
R1 Address of the parameter list, or equal to R15 if no parameters are passed
R2 Undefined
R13 Caller's standard register save area
R14 Return address
R15 Entry point address

On entry into the assembler subroutine, these registers must contain the following values:
R0 Reserved
R1 Address of the parameter list, or zero
R12 Common anchor area (CAA) address
R13 Caller's DSA
R14 Return address
R15 Entry point address
All others Undefined

On entry into an assembler routine, the caller's registers (R14 through R12) are saved into the DSA provided by the caller. After allocating a DSA (which sets the NAB field correctly in the new DSA), the first halfword of the DSA is set to zero and the backchain is set appropriately.

At all times while the assembler routine is running, R13 must contain the executing routine's DSA.

At call and return points, R12 must contain the CAA address.

On exit from the assembler routine, these registers contain:
R0 Undefined
R1 Undefined
R14 Undefined
R15 Undefined
All others The contents they had upon entry

Considerations for Coding or Running Assembler Routines

This section summarizes some areas you might need to consider when coding or running an assembler routine under LE/VSE.

Condition Handling

LE/VSE default condition handling actions occur for assembler routines unless you have registered a user-written condition handler using CEEHDLR (see *LE/VSE Programming Reference* for more information about CEEHDLR).

LE/VSE relinquishes all enclave-level resources that were obtained by LE/VSE when the enclave terminates, and all process-level resources when the process terminates.

Access to the Inbound Parameter String

You can access the standardized form of the inbound parameter list for the assembler main routine any time after routine initialization by using one of the following:

- The CEE5PRM (query parameter string) callable service described in *LE/VSE Programming Reference*.

What CEE5PRM returns depends on your execution environment, and the run-time or compile-time options you specify. See "What the Enclave Returns from CEE5PRM" on page 396 for more information.
- The PARMREG output value from the CEEENTRY macro described in "CEEENTRY Macro— Generate an LE/VSE-Conforming Prolog" on page 341.

Overlay Programs

LE/VSE does not provide explicit support for overlay programs. If programs are overlaid, LE/VSE imposes the following restrictions:

- All LE/VSE routines and static data must be placed in the root segment.
- All named routines and static data referred to by LE/VSE must be in the root segment.
- All ENTRY values or static data addresses passed to any LE/VSE service must point to routines in the root segment.
- All routines in the save area chain must be in storage for the whole time that they are in the chain.
- Calls that cause a new overlay segment to be loaded must not result in the calling segment being overlaid.
- Calls that cause a new overlay segment to be loaded must be between two routines in the same language (that is, they cannot be ILC calls).
- LE/VSE-conforming COBOL and PL/I routines cannot contain calls to the SORT program. The LE/VSE sort interface does not support overlay programs.

CEESTART, CEEMAIN, and CEEFMAIN

Assembler programs cannot call or use directly CEESTART, CEEMAIN, or CEEFMAIN as a standard entry point. Results are unpredictable if this rule is violated.

LE/VSE Library Routine Retention

LE/VSE library routine retention is a function that provides a performance improvement for those applications running in the batch environment with the following attributes:

- The application invokes programs that require LE/VSE.
- The application is not LE/VSE-conforming. That is, LE/VSE is not already initialized when the application invokes programs that require LE/VSE.
- The application repeatedly invokes programs that require LE/VSE.
- The application or subsystem is not using LE/VSE preinitialization services.

Note: LE/VSE library routine retention is not supported under CICS.

The use of library routine retention does not affect the behavior of applications other than improving their performance.

LE/VSE provides a macro called CEELRR, which is used in an assembler program to initialize library routine retention and to terminate library routine retention. See "CEELRR Macro— Initialize/Terminate LE/VSE Library Routine Retention" on page 339 for details about the CEELRR macro.

In addition, LE/VSE provides two sample programs that use the CEELRR macro:

CEELRRIN
This routine uses the CEELRR macro to initialize library routine retention. If LE/VSE is installed in the default sublibraries, you can find the source for this routine in member CEELRRIN.A in the PRD2.SCEEBASE sublibrary. The object module containing CEELRRIN can also be found in PRD2.SCEEBASE.

CEELRRTR
This routine uses the CEELRR macro to terminate library routine retention. If LE/VSE is installed in the default sublibraries, you can find the source for this routine in member CEELRRTR.A in the PRD2.SCEEBASE sublibrary. The object module containing CEELRRTR can also be found in PRD2.SCEEBASE.

When an application initializes library routine retention, LE/VSE keeps a subset of its resources in memory after the environment terminates. As a result, subsequent invocations of LE/VSE-conforming programs within the application are much faster because the resources can be reused without having to be reacquired and reinitialized.

When library routine retention has been initialized, the resources that LE/VSE keeps in memory when it terminates include the following:

- LE/VSE run-time library routines
- LE/VSE storage associated with the management of the run-time library routines
- LE/VSE storage for startup control blocks

When library routine retention is terminated, the resources that LE/VSE kept in memory are freed. (Library routines are deleted and storage is freed.)

Note: If library routine retention is initialized, and the job step in which it is being used is terminated, the operating system frees the LE/VSE resources as part of job step termination.

Using Library Routine Retention

If you are going to use library routine retention, you need to be aware of the following:

- Library routine retention can only be used in the batch environment, not under CICS.
- In order to successfully initialize library routine retention or terminate library routine retention, LE/VSE must not be currently initialized.

  For example, if you use CEELRR with ACTION=INIT in an LE/VSE-conforming assembler program, library routine retention is not initialized, because the invocation of the assembler program caused LE/VSE to be initialized.

Library Routine Retention and Preinitialization

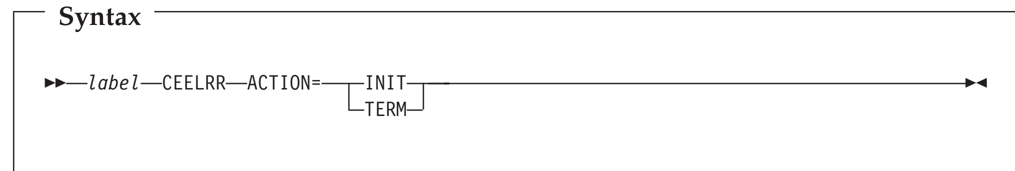
The LE/VSE preinitialization services can be used while library routine retention is initialized. However, the LE/VSE resources initialized and terminated with LE/VSE preinitialization services are not kept in memory when library routine retention is initialized. There is no sharing of resources between LE/VSE when initialized with preinitialization services and an environment initialized by invoking an HLL program without using preinitialization services. There is no

performance benefit of library routine retention for those applications that bring up an LE/VSE preinitialized environment, and then use the preinitialization services to invoke programs that require LE/VSE.

CEELRR Macro— Initialize/Terminate LE/VSE Library Routine Retention

CEELRR is used to tell LE/VSE to initialize and terminate library routine retention. The macro generates reentrant code.

Syntax



label
Assembler label on this macro generation.

ACTION=
The action to be performed by LE/VSE with regard to library routine retention. Valid values are INIT and TERM. A value of INIT tells LE/VSE to initialize library routine retention. A value of TERM tells LE/VSE to terminate library routine retention. You must specify the ACTION value.

Usage Notes:

1. The macro must be used in an assembler routine that is **not** LE/VSE-conforming.
2. The contents of the following registers are destroyed by the macro invocation:
   R14
   R15: Upon return, contains the return code
   R0
   R1
3. The code generated by the macro expansion assumes that R13 has a standard register save area (RSA) available.
4. One of the following return codes is put in R15 upon completion of the code generated by the CEELRR macro with ACTION=INIT:

   0 Library routine retention was successfully initialized.

   4 Library routine retention is already initialized. No action was taken.

   8 Library routine retention was not initialized. The parameter list is not recognized.

   12 Library routine retention was not initialized due to one of the following problems:
   - There was insufficient storage.
   - There was an error in an attempt to load CEEAEXT, CEEBLRR, CEEBINIT, or CEEBLIBM.

   16 Library routine retention was not initialized because LE/VSE is currently initialized.

   This return code can occur in the following example scenarios:
   - A program that is running with LE/VSE calls an assembler program that uses CEELRR with ACTION=INIT.

- An assembler program calls IGZERRE to initialize a reusable environment, and then it uses CEELRR with ACTION=INIT.
- A reusable environment is established with the RTEREUS run-time option and a call is made to an assembler program that uses CEELRR with ACTION=INIT.

20 Library routine retention was not initialized because the LE/VSE preinitialized environment has been established and is dormant.

This return code can occur when an assembler program calls CEEPIPI to preinitialize LE/VSE, and then it uses CEELRR with ACTION=INIT.

5. One of the following return codes is put in R15 upon completion of the code generated by the CEELRR macro with ACTION=TERM:

0 Library routine retention was successfully terminated. All resources associated with library routine retention were freed.

4 Library routine retention is not initialized. No action was taken.

8 Library routine retention was not terminated. The parameter list is not recognized.

12 Library routine retention was not terminated because there was an error in an attempt to load CEEAEXT or CEEBLRR.

16 Library routine retention was not terminated because LE/VSE is currently initialized.

This return code can occur in the following example scenarios:

- A program that is running with LE/VSE calls an assembler program that uses CEELRR with ACTION=TERM.
- An assembler program calls IGZERRE with the initialize function, and then it uses CEELRR with ACTION=TERM.
- A reusable environment is established with the RTEREUS run-time option and a call is made to an assembler program that uses CEELRR with ACTION=TERM.

20 Library routine retention was not terminated because the LE/VSE preinitialized environment has been established and is dormant.

This return code can occur when an assembler program calls CEEPIPI to preinitialize LE/VSE, and then it uses CEELRR with ACTION=TERM.

Assembler Macros

LE/VSE provides the following macros to assist in the entry and exit of assembler routines, to map the CAA and DSA, and to generate the appropriate fields in the program prolog area (PPA):

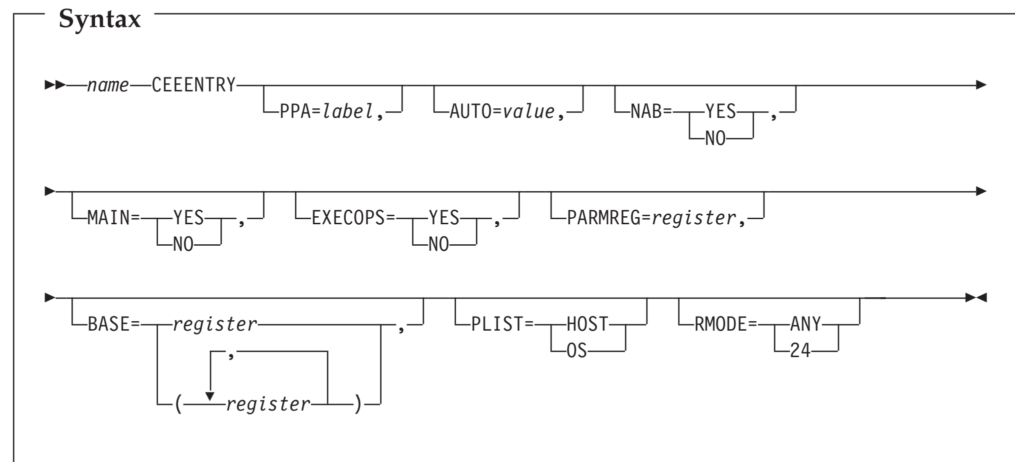
- CEEENTRY generates an LE/VSE-conforming prolog. You must use CEEENTRY in conjunction with the following macros, except for CEELOAD. (See page 341 for syntax.)
- CEETERM generates an LE/VSE-conforming epilog and terminates the assembler routine. (See page 343 for syntax.)
- CEECAA generates a CAA mapping. (See page 344 for syntax.)
- CEECIB generates a CIB mapping. (See page 345 for syntax.)
- CEEDSA generates a DSA mapping. (See page 345 for syntax.)
- CEEPPA generates the appropriate fields in the PPA in your assembler routine. The fields describe the entry point of an LE/VSE block. (See page 345 for syntax.)
- CEELOAD loads an LE/VSE-conforming routine, but without a corresponding service to later delete such routines. (See page 348 for syntax.)
- CEEFETCH loads an LE/VSE-conforming routine that can be later deleted using CEERELES. (See page 350 for syntax.)
- CEERELES deletes an LE/VSE-conforming routine that was loaded using CEEFETCH. (See page 353 for syntax.)

CEEENTRY Macro— Generate an LE/VSE-Conforming Prolog

CEEENTRY provides an LE/VSE-conforming prolog. Code is generated in cooperation with the CEEPPA macro (see "CEEPPA Macro— Generate a PPA" on page 345 for syntax).

You must use CEEENTRY in conjunction with the macros CEETERM, CEECAA, CEEDSA, and CEEPPA.

CEEENTRY assumes that the registers contain what is described in "Register Conventions" on page 335 for assembler main routines.



name
The entry name (and the CSECT name, if this is the first call to CEEENTRY).

PPA=

The *label* of the corresponding PPA (Program Prolog Area) generated using the CEEPPA macro. If unspecified, the name "PPA" is used.

AUTO=

The amount of space used by prolog code for the DSA and local automatic variables that are to be allocated for the duration of this routine. This *value* must be a multiple of doublewords. If unspecified, the size of the automatic area is the size of a DSA without any automatic variables. This is indicated by the label CEEDSASZ (the DSA mapping generated by the CEEDSA macro. See "CEEDSA Macro— Generate a DSA Mapping" on page 345 for syntax).

NAB=

YES

Indicates that the previous save area has an NAB (next available byte) value.

If you do not specify a value, **YES** is assumed.

In general,

- If your routine is always called by an LE/VSE-conforming assembler routine, specify NAB=YES.
- If your routine can be called by a non-LE/VSE-conforming assembler routine, specify NAB=NO.

NO

Indicates that the previous save area cannot contain the NAB. Code to find the NAB is generated. This parameter is ignored if MAIN=YES.

MAIN=

YES

Indicates that the LE/VSE environment should be brought up. **YES** designates this assembler routine as the main routine in the enclave.

If you do not specify a value, **YES** is assumed.

The following is accomplished by the macro invocation:

1. Save the caller's registers (R14 through R12) in the DSA provided by the caller.
2. Sets base register (see BASE).
3. Sets R12 with the address of CEECAA.
4. Sets R13 with the address of CEEDSA.
5. Sets PARMREG (R1 is the default) based on PLIST.
6. Register 0 and 2 are undefined.

NO

Designates this assembler routine as a subroutine in the enclave. **NO** should be specified when the LE/VSE environment is already active and only prolog code is needed.

The following is accomplished by the macro invocation:

1. Save the caller's registers (R14 through R12) in the DSA provided by the caller.
2. Sets base register (see BASE).
3. Sets R13 with the address of CEEDSA.
4. Sets PARMREG (see PARMREQ).
5. Register 0 is undefined.

EXECOPS=

YES

Indicates that the main routines are to honor run-time options on the inbound parameter string. This option is applicable only when **MAIN=YES** is in effect for the routine. The EXECOPS setting is ignored if **MAIN=NO** is specified.

If you do not specify a value, **YES** is assumed.

NO

Indicates that there are no run-time options in the inbound parameter string. LE/VSE considers the entire inbound parameter string as program arguments, but does not attempt to process run-time options and remove them from the inbound parameter string.

PARMREG=

Specifies the *register* to hold the inbound parameters. If you do not specify a value, Register **1** is assumed.

For MAIN=YES, the value of the PARMREG is determined based on PLIST. For MAIN=NO and PARMREG equal 1 (PARMREG defaults to 1), R1 is restored from the save area passed to the routine. For MAIN=NO and PARMREG not equal 1, R1 is used to load the specifed PARMREG.

BASE=

Establishes the base *register(s)* that you specify here, as the base register(s) for this module. If multiple base registers are required, they must be enclosed within brackets. If you do not specify a value, Register **11** is assumed. Register **12** should not be used.

If MAIN=YES is specified explicitly or by default, register 2 cannot be used.

PLIST=

Indicates that the main routines are to honor PLIST format on the inbound parameter string. This option is applicable only when **MAIN=YES** is in effect for the routine. The PLIST settings are ignored if **MAIN=NO** is specified. If you do not specify a value, **HOST** format is assumed.

The HOST format will set the specified PARMREG to the address of a field with a halfword-prefixed string of user parameters (run-time options have been removed). To obtain the inbound parameter list as specified, use PLIST=OS.

RMODE=

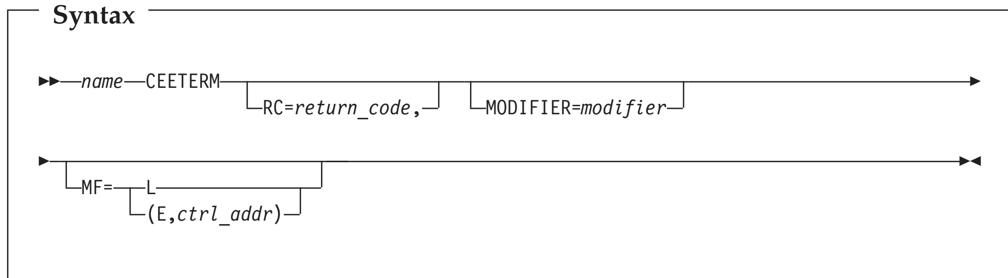
Allows the specification of the modules CSECT RMODE setting. All LE-enabled Assembler programs CSECTs are AMODE ANY. Valid settings for this option are **ANY** and **24**.

Usage Notes: Macro CEEENTRY allows multiple invocations within a single Assembly, but with the following restrictions:

- R15 MUST be pointing to the entry point for each CEEENTRY **MAIN=NO** section when invoked.
- Only a SINGLE **MAIN=YES** may be used in a single Assembly with multiple **MAIN=NO** CEEENTRY iterations.
- Use of **MAIN=NO** implies no CEETERM MF=L is allowed.

CEETERM Macro— Terminate an LE/VSE-Conforming Routine

CEETERM provides an LE/VSE-conforming epilog and is used to terminate, or return from, an LE/VSE-conforming routine. If used with a main entry, the appropriate call is made to LE/VSE termination routines.



name
The entry name (and the CSECT name, if this is for a main entry).

RC=
The return code that is to be placed into R15 after the *modifier* is added to it, if terminating a main routine. If returning from an LE/VSE subroutine, *return_code* itself is placed into R15, without *modifier* being added to it. You can specify *return_code* as a fixed constant, a variable, or register 2–12.

MODIFIER=
The return code *modifier* that is multiplied by the appropriate value (based upon the operating system), added to the return code and placed into R15, if terminating a main routine. The MODIFIER is independently placed into R0. You can specify *modifier* as a fixed constant, a variable, or register 2–12.

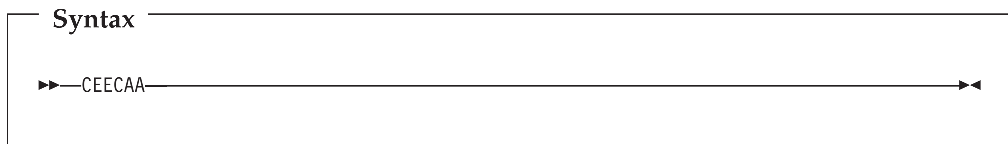
MF=L
Indicates the list form of the macro. A remote control program parameter list for the macro is defined, but the service is not invoked. The list form of the macro is usually used in conjunction with the execute form of the macro.

MF=(E,ctrl_addr)
Indicates the execute form of the macro. The service is invoked using the remote control program parameter list addressed by *ctrl_addr* (normally defined by the list form of the macro, it cannot be register 0).

Usage Notes:

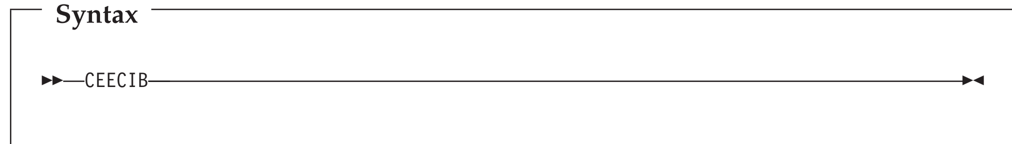
1. The MF=L and the MF=(E,ctrl_addr) parameters cannot both be coded for the same macro invocation. If neither is coded, the immediate form of the macro is used. The immediate form generates an inline parameter list, and generates nonreentrant code.
2. The address of the name can be specified as a register using parentheses ().
3. The macro invocation destroys the following registers:
   R1
   R14
   R15

CEECAA Macro— Generate a CAA Mapping



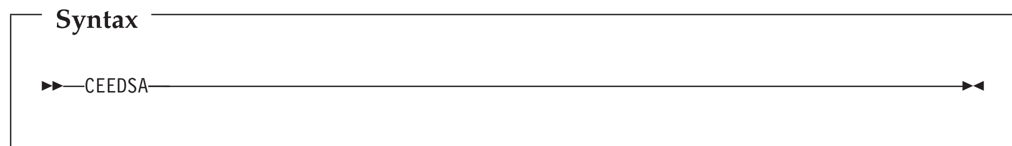
CEECAA is used to generate a common anchor area (CAA) mapping. This macro has no parameters, and no label can be specified. CEECAA is required for the CEEENTRY macro.

CEECIB Macro— Generate a CIB Mapping



CEECIB is used to generate a a condition information block (CIB) mapping. This macro has no parameters, and no label can be specified. CEECIB is required for condition handling and debugging purposes. For details, refer to the section "Debugging with the Condition Information Block" in the *LE/VSE Debugging Guide and Run-Time Messages*.

CEEDSA Macro— Generate a DSA Mapping

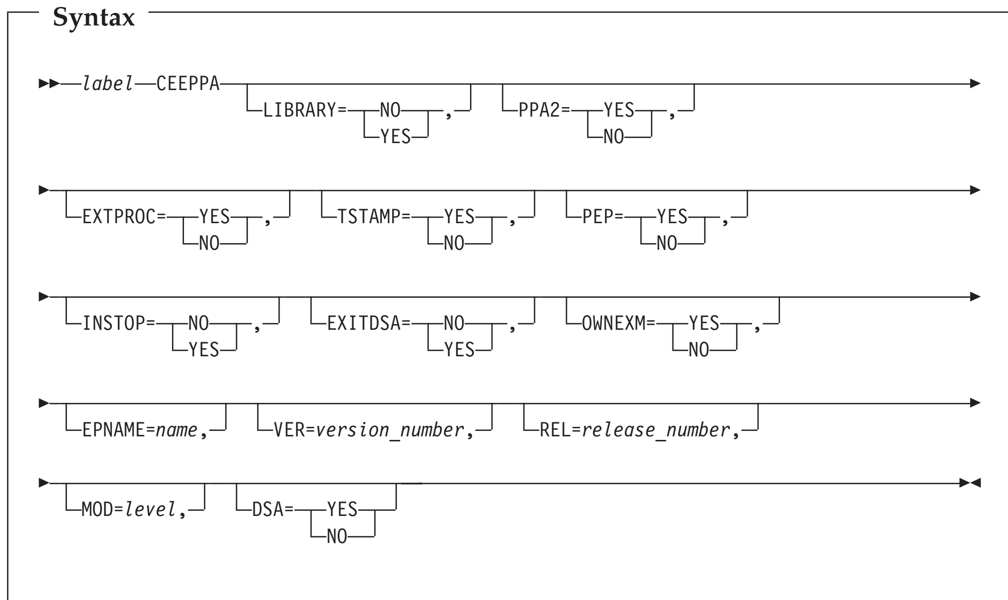


CEEDSA is used to generate a dynamic save area (DSA) mapping. This macro has no parameters, and no label can be specified. The minimum size of the DSA is specified by the assembler equated symbol CEEDSASZ. CEEDSA is required for the CEEENTRY macro.

CEEPPA Macro— Generate a PPA

CEEPPA is used to generate the LE/VSE program prolog area (PPA). The PPA defines constants that describe the entry point of an LE/VSE block. It is generated at the time of assembly; one PPA is generated per entry point.

The CEEPPA macro is required for the CEEENTRY macro.



label
: The name of the PPA.

  If you specified a name for **PPA** in the CEEENTRY macro, you must specify the same name here. If you did not specify a name for **PPA** in the CEEENTRY macro, you must specify “PPA” (the CEEENTRY default PPA label) as the name here.

LIBRARY=
: Indicates whether the routine is an LE/VSE library routine. Valid values for LIBRARY are **YES** and **NO**. If you do not specify a value, **NO** is used. Use of this IBM-supplied default is recommended.

PPA2=
: Instructs the macro to generate a PPA2 or suppress the generation of the PPA2. A PPA2 is a program prolog area that defines constants for the CSECT. Only one is used, independent of the number of entry points.

  Valid values for PPA2 are **YES** and **NO**. If you do not specify a value, **YES** is used, which generates a PPA2 field.

EXTPROC=
: Indicates whether this routine is an external procedure or an internal procedure. An internal procedure is known only within the CSECT and can be called only from within the CSECT.

  Valid values for EXTPROC are **YES** and **NO**. If you do not specify a value, **YES** is used, which indicates that the block is an external procedure.

TSTAMP=
: Indicates whether a timestamp, indicating the date and time of assembly, should be generated. Valid values for TSTAMP are **YES** and **NO**. If you do not specify a value, **YES** is used and a timestamp is generated.

PEP=
: Indicates whether this entry point is primary or secondary. A secondary entry point is an alternate entry point. Some LE/VSE facilities, such as CEE5DMP, report information based on the primary entry point only.

Valid values for PEP are **YES** and **NO**. If you do not specify a value, **YES** is used, which indicates that this is a primary entry point (PEP).

INSTOP=
Indicates whether time spent in this routine should be attributed to the program (rather than to the "system"). Valid values for INSTOP are **YES** and **NO**. If you do not specify a value, **NO** is used, which indicates that time should be attributed to the system.

EXITDSA=
Indicates whether the code should gain control on GOTO out of block. Valid values for EXITDSA are **YES** and **NO**. If you do not specify a value, **NO** is used, which indicates that the code does not gain control for GOTO out of block. Use of this IBM-supplied default is recommended.

OWNEXM=
Specifies whether this routine should participate in condition handling according to the member id set in ID. Valid values for OWNEXM are **YES** and **NO**. If you do not specify a value, **YES** is used, which indicates that participation is desired. Use of this IBM-supplied default is recommended.

EPNAME=
Indicates the entry point *name*. If you do not specify a value, the name of the CSECT is used.

VER=
The *version number* for the routine. This field is not interrogated by LE/VSE. Valid values for VER are **1** through **99**. If you do not specify a value, **1** is used.

REL=
The *release number* for the routine. This field is not interrogated by LE/VSE. Valid values for REL are **1** through **99**. If you do not specify a value, **1** is used.

MOD=
The modification *level* for the routine. This field is not interrogated by LE/VSE. Valid values for MOD are **1** through **99**. If you do not specify a value, **0** is used.

DSA=YES
Indicates whether this procedure has a DSA. Valid values for DSA are **YES** and **NO**. If you do not specify a value, **YES** is used, which indicates that the code does have an associated DSA. Use of this IBM-supplied default is recommended.

CEELOAD Macro— Dynamically Load a Routine

CEELOAD is used to dynamically load an *LE/VSE-conforming* routine. It does not create a nested enclave, so the target of CEELOAD must be a subroutine.

Notes:

1. If you use CEELOAD, you should be aware that there is no corresponding service to delete LE/VSE-conforming routines. You should not use system services to delete phases that you load using CEELOAD; during thread (if SCOPE=THREAD) or enclave (if SCOPE=ENCLAVE) termination, LE/VSE deletes phases loaded by CEELOAD.
2. If you wish to dynamically load LE/VSE-conforming routines and have the possibility to later delete these routines, you should use CEEFETCH instead of CEELOAD. For details, see "CEEFETCH Macro— Dynamically Load a Routine that Can Be Later Deleted" on page 350.

Using CEELOAD imposes restrictions on further dynamic loading or dynamic calls or fetches:

- You cannot dynamically load a routine with CEELOAD that has already been dynamically loaded by CEELOAD or has been fetched or dynamically called.
- You cannot fetch or dynamically call a routine that has already been dynamically loaded by CEELOAD.

Results are unpredictable if these rules are violated.

If CEELOAD completes successfully, the address of the loaded routine is found in R15. You can then invoke the routine using BALR 14,15 (or BASSM 14,15).

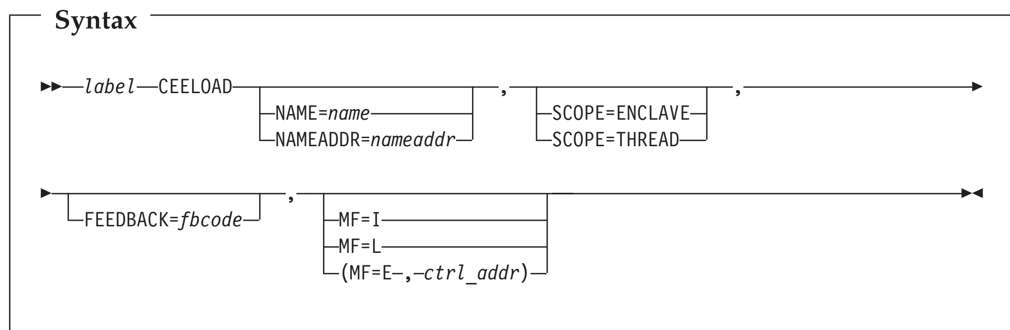
LE/VSE returns the address of the target routine with the high-order bit indicating the addressing mode (AMODE) of the routine. LE/VSE-enabled programs return in the AMODE in which they are entered. Because LE/VSE does not provide any AMODE switching on behalf of the target routine, you must provide any necessary AMODE switching code.

The macro invocation destroys the following registers:

R0
R1
R14
R15 (upon return, contains the target address)

When the macro code is expanded and run, the following assumptions are made:

- R12 points to the CAA.
- R13 has a standard LE/VSE DSA available.



label
The assembler label you give to this invocation of the macro. A label is required if MF=L; otherwise it is optional.

NAME=
The name of the entry point to be loaded by LE/VSE. If MF=I, you must specify either NAME or NAMEADDR *but not both.* If MF=L or MF=E, NAME is optional: however if you *do* specify NAME, you *must not* specify NAMEADDR.

NAMEADDR=
The address of a halfword-prefixed name that should be loaded by LE/VSE. This can be an A-type address or a register (registers 2 to 11). If MF=I, you must specify either NAME or NAMEADDR *but not both.* If MF=L or MF=E, NAMEADDR is optional: however if you *do* specify NAMEADDR, you *must not* specify NAME.

SCOPE=THREAD
Indicates that the load is to be scoped to the thread level. Phases loaded at the thread level are deleted automatically at thread termination.

SCOPE=ENCLAVE
Indicates that the load is to be scoped to the enclave level. Phases loaded at the enclave level are deleted automatically at enclave termination.

If neither SCOPE=ENCLAVE nor SCOPE=THREAD is specified, SCOPE=ENCLAVE is used.

FEEDBACK=
The name of a variable to contain the resulting 12-byte feedback token. If you omit this parameter, any nonzero feedback token that results is signaled.

The following symbolic conditions might be returned from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE38M	3	3350	Unable to find the event handler.
CEE38N	3	3351	Unable to properly initialize the event handler.
CEE39K	1	3380	The target phase was not recognized by Language Environment.
CEE3DC	3	3500	Not enough storage was available to load *phase-name*.
CEE3DD	3	3501	The phase *phase-name* was not found.
CEE3DE	3	3502	The phase name *phase-name* was too long.
CEE3DF	3	3503	The load request for phase *phase-name* was unsuccessful.

MF=I
Indicates the immediate form of the macro. The immediate form generates an inline parameter list, and generates nonreentrant code.

MF=L
Indicates the list form of the macro. A remote control program parameter list for the macro is defined, but the service is not invoked. The list form of the macro is usually used in conjunction with the execute form of the macro.

MF=(E,ctrl_addr)
Indicates the execute form of the macro. The service is invoked using the remote control program parameter list addressed by *ctrl_addr* (normally defined by the list form of the macro).

Usage Notes

- LE/VSE issues the appropriate operating system load command according to LE/VSE search order (described in “Specifying the Search Order” on page 29), and performs the necessary dynamic updates to accommodate the new phase.
- There is no corresponding service to delete LE/VSE-conforming assembler routines. You should not use system services to delete phases that you load using CEELOAD.
- LE/VSE performs any language-related initialization that is required.
- You cannot use CEELOAD to load C modules that use writable static. C programs that are compiled as RENT will have writable static, C programs that are compiled as NORENT do not have writable static. If any program in a phase uses writable static, the PHASE will require writable static switching. CEELOAD does not provide support to switch writable static when calling a C function. To circumvent this restriction, if possible, compile your C modules using the NORENT option.

CEEFETCH Macro— Dynamically Load a Routine that Can Be Later Deleted

CEEFETCH is used to dynamically load an *LE/VSE-conforming* routine that may be later deleted. Because CEEFETCH does not create a nested enclave, the target of CEEFETCH must be a subroutine.

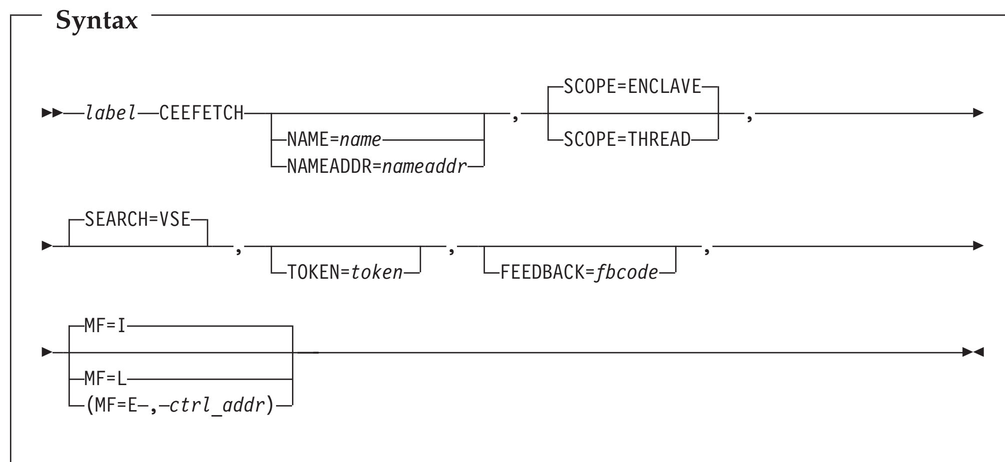
Use the CEERELES macro to delete routines loaded with CEEFETCH. You should not use system services to delete modules that you load using CEEFETCH, since LE/VSE deletes modules loaded by CEEFETCH during either:
- thread (if SCOPE=THREAD) termination
- enclave (if SCOPE=ENCLAVE) termination

If CEEFETCH completes successfully, the address of the target routine is found in R15. You can then invoke the routine using the BALR 14,15 (or BASSM 14,15) instruction.

LE/VSE returns the address of the target routine with the high-order bit indicating the addressing mode (AMODE) of the routine. LE/VSE-enabled programs return in the AMODE in which they are entered. Because LE/VSE does not provide any AMODE switching on behalf of the target routine, you must provide any necessary AMODE switching code.

The macro invocation destroys the following registers:
R0
R1
R14
R15 (upon return, contains the target address)

When the macro code is expanded and run, the following assumptions are made:
- R12 points to the CAA.
- R13 has a standard LE/VSE DSA available.



label
: The assembler label you give to this invocation of the macro. A label is required if MF=L; otherwise it is optional.

NAME=*name*
: The name of the entry point to be loaded by LE/VSE. The maximum length of name is eight characters. You cannot specify NAME and NAMEADDR together.

NAMEADDR=*nameaddr*
: The address of a halfword-prefixed name that should be loaded by LE/VSE. A halfword prefix name is a string where the first two bytes identify the length of a name string and are followed by the name string itself. This can be an A-type address or a register (register 2 through 11). The address of the name can be specified as a register using parentheses (). The maximum length of the name is 8 characters. You cannot specify NAME and NAMEADDR together.

SCOPE=THREAD
: Indicates that the load is to be scoped to the thread level. Modules loaded at the thread level are deleted automatically at thread termination.

SCOPE=ENCLAVE
: Indicates that the load is to be scoped to the enclave level. Modules loaded at the enclave level are deleted automatically at enclave termination; this is the default.

TOKEN=*token*
: The name of a variable to contain the resulting 4-byte token. This variable must be passed to the CEERELES macro if the load module is to deleted. If MF=I or MF=L are specified, you must specify TOKEN.

SEARCH=VSE
: The SEARCH option is accepted for LE z/OS compatibility only. If this option is set to anything other than VSE, it is accepted but ignored at execution time. An informational message is produced in the assembler listing when this option is specified and set to anything other than VSE.

FEEDBACK=*fbcode*
: The name of a variable to contain the resulting 12-byte feedback token. If you omit this parameter, any nonzero feedback token that results is signaled.

The following symbolic conditions might be returned from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE38M	3	3350	CEE5ADM or CEE5MBR could not find the event handler.
CEE38N	3	3351	CEE5ADM or CEE5MBR could not properly initialize the event handler.
CEE39K	1	3380	The target load module was not recognized by Language Environment.
CEE3DC	3	3500	Not enough storage was available to load *module-name*.
CEE3DD	3	3501	The module *module-name* was not found.
CEE3DE	3	3502	The module name *module-name* was too long.
CEE3DF	3	3503	The load request for module *module-name* was unsuccessful.
CEE3N9	2	3817	The member event handler did not return a usable function pointer.

MF=I
Indicates the immediate form of the macro. The immediate form generates an inline parameter list, and generates nonreentrant code. This is the default value.

MF=L
Indicates the list form of the macro. A remote control program parameter list for the macro is defined, but the service is not invoked. The list form of the macro is usually used in conjunction with the execute form of the macro.

MF=(E,ctrl_addr)
Indicates the execute form of the macro. The service is invoked using the remote control program parameter list addressed by *ctrl_addr* (usually defined by the list form of the macro).

Only one of the MF=I, MF=L, or MF=(E, ctrl_addr) parameters can be coded for the same macro invocation. If none is coded, the immediate form of the macro is used.

Usage Notes

1. LE/VSE issues the appropriate operating system load command, and performs the necessary dynamic updates to accommodate the target load module.
2. LE/VSE performs any language-related initialization required.
3. Any C or PL/I module that will be fetched, dynamically called, or CEEFETCHed more than once must be reentrant.
4. CEEFETCH is only supported by LE/VSE for:
   - FETCHABLE reentrant C or PL/I subroutines.
   - Reentrant COBOL/VSE programs.

   CEEFETCH only supports COBOL load modules that are constructed entirely of COBOL programs compiled with an LE/VSE-conforming compiler (COBOL/VSE), and link-edited with LE/VSE.
5. If a PL/I reentrant fetchable subroutine is to be the entry point of a target CEEFETCHed load module, the following rules must be followed or unpredictable results may occur:

- The main environment must be created by a C routine or assembler program.
- Any CEEFETCHed PL/I subroutines must be reentrant, fetchable and in AMODE(31).
- To ensure the PL/I-specific run-time is available at the time a PL/I routine is executed, one of the following must be true in the executable program:
  - A CEEFETCHed C subroutine directly calls the PL/I for VSE subroutine.
  - A PL/I for VSE subroutine is present in the main executable program.
  - The LE/VSE PL/I signature CSECT CEESG010 is explicitly included in the main executable program.

CEERELES Macro— Dynamically Delete a Routine

CEERELES is used to dynamically delete an *LE/VSE-conforming* routine.

The macro invocation destroys the following registers:
R0
R1
R14
R15

When the macro code is expanded and run, the following assumptions are made:
- R12 points to the CAA.
- R13 has a standard LE/VSE DSA available.

Syntax

```
►►──label──CEERELES──┬─────────────┬──,──┬─────────────────┬──,────────────►
                     └─TOKEN=token─┘     └─FEEDBACK=fbcode─┘

   ┌─MF=I──────────────────┐
►──┼───────────────────────┼───────────────────────────────────────────────►◄
   ├─MF=L──────────────────┤
   └─(MF=E─,─ctrl_addr)────┘
```

label
: The assembler label you give to this invocation of the macro. A label is required if MF=L; otherwise it is optional.

TOKEN=*token*
: The name of a variable that contains the token returned by the CEEFETCH macro. If MF=I or MF=L are specified, you must specify TOKEN.

FEEDBACK=*fbcode*
: The name of a variable to contain the resulting 12-byte feedback token. If you omit this parameter, any nonzero feedback token that results is signaled.

The following symbolic conditions might be returned from this service:

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE000	0	—	The service completed successfully.
CEE38N	3	3351	An event handler was unable to process the request successfully.

Symbolic Feedback Code	Severity	Message Number	Message Text
CEE39K	1	3380	The target load module was not recognized by Language Environment.
CEE3DG	3	3504	Delete service request for module-name was unsuccessful.
CEE3E0	3	3520	The token passed to the CEERELES macro was invalid.

MF=I
Indicates the immediate form of the macro. The immediate form generates an inline parameter list, and generates nonreentrant code. This is the default value.

MF=L
Indicates the list form of the macro. A remote control program parameter list for the macro is defined, but the service is not invoked. The list form of the macro is usually used in conjunction with the execute form of the macro.

MF=(E,ctrl_addr)
Indicates the execute form of the macro. The service is invoked using the remote control program parameter list addressed by *ctrl_addr* (usually defined by the list form of the macro).

Only one of the MF=I, MF=L, or MF=(E, ctrl_addr) parameters can be coded for the same macro invocation. If none is coded, the immediate form of the macro is used.

Usage Notes

1. LE/VSE issues the appropriate operating system delete command, and performs the necessary dynamic updates to accommodate the deleted load module.
2. LE/VSE performs any language-related cleanup required.
3. LE/VSE only supports CEERELES for programs that are:
   - FETCHABLE reentrant C or PL/I subroutines.
   - Reentrant COBOL/VSE programs.

   CEERELES only supports COBOL load modules that are constructed entirely of COBOL programs compiled with an LE/VSE-conforming compiler (COBOL/VSE), and link-edited with LE/VSE.

Example of Assembler Main Routine

The following shows a simple assembler main routine. In the example, the LE/VSE environment is established, a message showing control is received in the routine, and the LE/VSE environment terminates with a zero return code passed in R15 to the invoker.

If you write an assembler main routine, nominate the routine as the phase entry point using the END statement, as shown in the following example. Otherwise, you must explicitly declare the routine as the entry point at link-edit time.

```
*COMPILATION UNIT: LEASMMN
* ===================================================================
*
*        A simple main assembler routine that brings up the
*        LE/VSE environment, prints a message in the main routine,
*        and returns with a return code of 0, modifier of 0.
*
* ===================================================================
MAIN     CEEENTRY PPA=MAINPPA
*
*        Invoke CEEMOUT to issue a message for us
*
         CALL  CEEMOUT,(STRING,DEST,0)      Omitted feedback code
*
*   Terminate the LE/VSE environment and return to the caller
*
         CEETERM  RC=0,MODIFIER=0
* ====================================================================
*              CONSTANTS AND WORKAREAS
* ====================================================================
*
DEST     DC    F'2'
STRING   DC    Y(STRLEN)
STRBEGIN DC    C'In the main routine'
STRLEN   EQU   *-STRBEGIN
MAINPPA  CEEPPA  ,                Constants describing the code block
         CEEDSA  ,                Mapping of the dynamic save area
         CEECAA  ,                Mapping of the common anchor area
         END   MAIN               Nominate MAIN as the entry point
```

Figure 119. Example of Simple Main Assembler Routine

Example of an Assembler Main Calling an Assembler Subroutine

Figure 120 illustrates a simple assembler main routine that calls the DISPARM subroutine shown in Figure 121 on page 357.

```
*COMPILATION UNIT: LEASMSB
* ====================================================================
*       A simple main assembler routine brings up LE/VSE, calls a
*       subroutine, and returns with a return code of 0.
* ====================================================================
SUBXMP   CEEENTRY PPA=XMPPPA,AUTO=WORKSIZE
         USING WORKAREA,R13
* -------------------------------------------------------------------
* Invoke CEEMOUT to issue the greeting message
*
         CALL  CEEMOUT,(HELLOMSG,DEST,FBCODE),VL,MF=(E,CALLMOUT)
*
* No plist to DISPARM, so zero R1. Then call it.
*
         SR    R01,R01
         CALL  DISPARM
*
* Invoke CEEMOUT to issue the farewell message
*
         CALL  CEEMOUT,(BYEMSG,DEST,FBCODE),VL,MF=(E,CALLMOUT)
*
* Terminate Language Environment and return to the caller
*
         CEETERM  RC=0
* ===================================================================
*              CONSTANTS
* ===================================================================
*
HELLOMSG DC    Y(HELLOEND-HELLOSTR)
HELLOSTR DC    C'Hello from the sub example.'
HELLOEND EQU   *
*
BYEMSG   DC    Y(BYEEND-BYESTART)
BYESTART DC    C'Terminating the sub example.'
BYEEND   EQU   *
*
DEST     DC    F'2'               Destination is the LE message file
*
XMPPPA   CEEPPA  ,                Constants describing the code block
* ===================================================================
*        The Workarea and DSA
* ===================================================================
WORKAREA DSECT
         ORG   *+CEEDSASZ         Leave space for the DSA fixed part
CALLMOUT CALL  ,(,,),VL,MF=L      3-argument parameter list
*
FBCODE   DS    3F                 Space for a 12-byte feedback code
*
*
         DS    0D
WORKSIZE EQU   *-WORKAREA
         CEEDSA  ,                Mapping of the dynamic save area
         CEECAA  ,                Mapping of the common anchor area
*
R01      EQU   1
R13      EQU   13
         END   SUBXMP             Nominate SUBXMP as the entry point
```

Figure 120. Example of an Assembler Main Routine Calling a Subroutine

```
*COMPILATION UNIT:  LEASMPRM
* ====================================================================
*
*        Shows an assembler subroutine that displays inbound
*        parameters and returns.
*
* ====================================================================
DISPARM  CEEENTRY PPA=PARMPPA,AUTO=WORKSIZE,MAIN=NO
         USING WORKAREA,R13
*
* --------------------------------------------------------------------
*
* Invoke CEE5PRM to retrieve the command parameters for us
*
         CALL  CEE5PRM,(CHARPARM,FBCODE),VL,MF=(E,CALL5PRM)
*
* Check the feedback code from CEE5PRM to see if everything worked.
*
         CLC   FBCODE(8),CEE000
         BE    GOT_PARM
*
* Invoke CEEMOUT to issue the error message for us
*
         CALL  CEEMOUT,(BADFBC,DEST,FBCODE),VL,MF=(E,CALLMOUT)
         B     GO_HOME                 Time to go....
*
GOT_PARM DS    0H
*
* See if the parm string is blank.
*
         CLC   CHARPARM(80),=CL80' '   Is the parm empty?
         BNE   DISPLAY_PARM            No. Print it out.
*
* Invoke CEEMOUT to issue the error message for us
*
         CALL  CEEMOUT,(NOPARM,DEST,FBCODE),VL,MF=(E,CALLMOUT)
         B     GO_HOME                 Time to go....
*
DISPLAY_PARM   DS  0H
*
* Set up the plist to CEEMOUT to display the parm.
*
         LA    R02,80        Get the size of the string
         STH   R02,BUFFSIZE  Save it for the len-prefixed string
*
* Invoke CEEMOUT to display the parm string for us
*
         CALL  CEEMOUT,(BUFFSIZE,DEST,FBCODE),VL,MF=(E,CALLMOUT)
*
* Return to the caller
*
GO_HOME  DS    0H
         CEETERM  RC=0
```

Figure 121. Example of a Called Assembler Subroutine (Part 1 of 2)

```
* ====================================================================
*             CONSTANTS
* ====================================================================
*
DEST     DC    F'2'              Destination is the LE message file
CEE000   DS    3F'0'             Success feedback code
*
BADFBC   DC    Y(BADFBEND-BADFBSTR)
BADFBSTR DC    C'Feedback code from CEE5PRM was nonzero.'
BADFBEND EQU   *
*
NOPARM   DC    Y(NOPRMEND-NOPRMSTR)
NOPRMSTR DC    C'No user parm was passed to the application.'
NOPRMEND EQU   *
*
*
PARMPPA  CEEPPA ,                Constants describing the code block
* ====================================================================
*        The Workarea and DSA
* ====================================================================
WORKAREA DSECT
         ORG   *+CEEDSASZ        Leave space for the DSA fixed part
*
CALL5PRM CALL  ,(,),VL,MF=L      2-argument parameter list
CALLMOUT CALL  ,(,,),VL,MF=L     3-argument parameter list
FBCODE   DS    3F                Space for a 12-byte feedback code
*
BUFFSIZE DS    H                 Halfword prefix for following string
CHARPARM DS    CL255             80-byte buffer
*
*
         DS    0D
WORKSIZE EQU   *-WORKAREA
         CEEDSA  ,               Mapping of the dynamic save area
         CEECAA  ,               Mapping of the common anchor area
*
R02      EQU   2
R13      EQU   13
         END
```

Figure 121. Example of a Called Assembler Subroutine (Part 2 of 2)

Invoking Callable Services from Assembler Routines

The interface to a callable service is the same as the interface described above for assembler routines. An example of calling the CEEGTST (Get Heap Storage) callable service is shown in the following sample.

An X'80000000' placed in the last parameter address slot indicates that the *fc* (feedback code) parameter is omitted.

```
*
*   R12 = A(CAA)
*   R13 = DSA
*   This example is non-reentrant.
*
         LA   R1,PLIST
         L    R15,=V(CEEGTST)
         BALR R14,R15
         . . . .
PLIST    DS   0D
         DC   A(HEAP_ID)
         DC   A(SIZE)
         DC   A(ADDR)
         DC   A(X'80000000)
HEAP_ID  DC   F'0'                   Heap id for the user
SIZE     DC   F'256'                 Size of storage to allocate
ADDR     DC   F'0'                   Address of allocated storage
```

Figure 122. Sample Invocation of a Callable Service from Assembler

System Services Available to Assembler Routines

LE/VSE provides a number of services that the host system typically provides. Each of these system-provided services belongs to one of three categories, depending on whether it can and ought to be used in LE/VSE:

- The system-provided service **can** be used, but you must manage the resource. Examples are ENQ and DEQ.
- The system-provided service **can**, but **should not** be used. The system-provided service might not have the desired effect. For example, instead of using GETVIS and FREEVIS, use the LE/VSE dynamic storage callable services.
- The system-provided service **must not** be used. If you use this service, it directly interferes with the LE/VSE environment. For example, any STXIT AB or STXIT PC that you issue interferes with LE/VSE condition handling.

Whenever possible, non-LE/VSE-conforming assembler routines should use the equivalent LE/VSE services. A list of the equivalent services is provided in Table 51.

Table 51. LE/VSE's Equivalent Host Services

Host service	LE/VSE Equivalent	Usability
ATTACH/DETACH/CHAP	No equivalent LE/VSE function.	LE/VSE does not provide specific support for multitasking. If you use these host services, you must manage the subtasks.
CANCEL	Call CEESGL with a severity 4 condition, call CEE5ABD, or have the assembler user exit request an abend at termination.	If you want LE/VSE to terminate the environment, this host service **must not** be used.

Table 51. LE/VSE's Equivalent Host Services (continued)

Host service	LE/VSE Equivalent	Usability
CDLOAD/CDDELETE	Use the LE/VSE CEELOAD assembler macro (see page 348).	If you are introducing a new language into the environment, host services **must not** be used. The new language is not properly initialized. If you are not introducing a new language into the environment, then host services can be used. However, you must manage the loaded routines.
DUMP	Call CEESGL with a severity 4 condition, call CEE5ABD, or have the assembler user exit request an abend at termination.	If you want LE/VSE to terminate the environment, this host service **must not** be used.
ENQ/DEQ	No equivalent LE/VSE function.	These services can be used.
EOJ	Use the LE/VSE CEETERM macro, or the LE/VSE-conforming language equivalent, from the main program in your application.	If you want LE/VSE to terminate the environment, this host service **must not** be used.
EXEC CICS GETMAIN/FREEMAIN	For automatic storage (block related), use LE/VSE's stack storage. For non-block related storage (that is, the storage persists beyond the current activation), use LE/VSE heap storage.	Host services can, but should not, be used. Use of equivalent LE/VSE storage management services is advised.
EXEC CICS HANDLE ABEND	Use LE/VSE's condition management callable services—CEEHDLR, CEEHDLU, and CEESGL.	Host services should not be used. They interfere with LE/VSE's condition management.
EXEC CICS LOAD/DELETE	No equivalent LE/VSE function.	Host services can be used, but you must manage the loaded routines.
EXEC CICS XCTL/LINK	No equivalent LE/VSE function.	These services can be used.
FETCH	Use the LE/VSE CEELOAD assembler macro (see page 348).	If you are introducing a new language into the environment, host services **must not** be used. The new language is not properly initialized. If you are not introducing a new language into the environment, then host services can be used. However, you must manage the loaded routines.
GETIME	Call LE/VSE date and time services.	This service can be used.
GETVIS/FREEVIS	For automatic storage (block related), use LE/VSE's stack storage. For non-block related storage (that is, the storage persists beyond the current activation), use LE/VSE heap storage.	Host services can, but should not, be used. Use of equivalent LE/VSE storage management services is advised.
JDUMP	Call CEESGL with a severity 4 condition, call CEE5ABD, or have the assembler user exit request an abend at termination.	If you want LE/VSE to terminate the environment, this host service **must not** be used.
LOAD	Use the LE/VSE CEELOAD assembler macro (see page 348).	If you are introducing a new language into the environment, host services **must not** be used. The new language is not properly initialized. If you are not introducing a new language into the environment, then host services can be used. However, you must manage the loaded routines.
OPEN/CLOSE GET/PUT READ/WRITE	No equivalent LE/VSE function.	These host services can be used.
PDUMP/SDUMP/SDUMPX	Call CEE5DMP.	These host services can be used.

Table 51. LE/VSE's Equivalent Host Services (continued)

Host service	LE/VSE Equivalent	Usability
SETIMER/TTIMER	No equivalent LE/VSE function.	These host services can be used.
SETT/TESTT	No equivalent LE/VSE function.	These host services can be used.
STXIT AB/PC	Use LE/VSE's condition management callable services—CEEHDLR, CEEHDLU, and CEESGL.	Host services **must not** be used. They interfere with LE/VSE's condition management.
STXIT IT/OC/TT	No equivalent LE/VSE function.	These host services can be used.
WAIT/WAITM/POST	No equivalent LE/VSE function.	Host services can be used.
WTO	Call CEEMOUT. This writes to the message file.	Host services can be used.

Chapter 27. Using Preinitialization Services

You can use preinitialization to enhance the performance of your application. Preinitialization lets an application initialize an HLL environment once, perform multiple executions using that environment, and then explicitly terminate the environment. Because the environment is initialized only once (even if you perform multiple executions), you free up system resources and allow for faster responses to your requests.

This chapter describes the LE/VSE-supplied routine, CEEPIPI, that provides the interface for preinitialized routines. Using CEEPIPI, you can initialize an environment, invoke applications, terminate an environment, and add an entry to the PIPI table. (The PIPI table contains the names and entry point addresses of routines that can be executed in the preinitialized environment.)

This chapter also describes reentrancy considerations for a preinitialized environment, user exit invocation, stop semantics, service routines, and an example of CEEPIPI invocation.

Note: The CEEPIPI preinitialization service is designed to call HLL Language Environment-enabled programs from a non-LE/VSE environment running in a batch partition. This service is a batch interface and cannot be used under CICS. Under CICS you should use EXEC CICS LINK/XCTL to introduce an LE/VSE-conforming HLL routine.

Understanding the Basics

From a non-LE/VSE-conforming driver (such as assembler) you can use LE/VSE preinitialization facilities to create and initialize a common run-time environment, execute applications written in an LE/VSE-conforming HLL multiple times within the preinitialized environment, and terminate the preinitialized environment. LE/VSE provides the CEEPIPI preinitialized interface to perform these tasks.

In the preinitialized environment, the first routine to execute can be treated as either the main routine or a subroutine of that execution instance. LE/VSE provides support for both of these types of preinitialized routines:
- Executing one main routine multiple times
- Executing subroutines multiple times

LE/VSE preinitialization is commonly used to enhance performance for repeated invocations of an application or for a complex application where there are many repetitive requests and where fast response is required. For instance, if an assembler routine invokes either a number of LE/VSE-conforming HLL routines or the same HLL routine a number of times, the creation and termination of that HLL environment multiple times is needlessly inefficient. A more efficient method is to create the HLL environment only once for use by all invocations of the routine.

You must be careful with regards to the currently-active PSW program mask settings, when non-LE conforming languages are involved in an application that:
1. Uses pre-initialisation services.
2. Are executed while the LE environment is inactive.

© Copyright IBM Corp. 1991, 2005

Each Language Environment-conforming language sets required program mask bits at initialisation. LE/VSE will maintain and control the correct program mask settings within the active environment. However, when the environment is dormant and non-LE conforming languages are executed, program mask bits will still be active for the Language Environment and may cause unexpected program interruptions. It is the programmers responsibility to set program mask bit settings for the non-LE conforming application and to restore the previous mask settings before activating the LE environment.

Compatibility

COBOL

LE/VSE honors the current COBOL interfaces to preinitialization, RTEREUS, ILBDSET0, and IGZERRE. For more information about these interfaces, see *IBM COBOL for VSE/ESA Migration Guide*

Using Preinitialization

The interface for preinitialized routines is a loadable routine called CEEPIPI. CEEPIPI is loaded as an RMODE(24) / AMODE(ANY) routine and returns in the AMODE of its caller when the request is satisfied.

CEEPIPI handles the requests and provides services for environment initialization, application invocation, and environment termination. All requests for services by CEEPIPI must be made from a non-LE/VSE environment. ("CEEPIPI Interface" on page 368 contains a detailed description and information about how to invoke each of these services.) The parameter list for CEEPIPI is an OS standard linkage parameter list. Each request to CEEPIPI is identified by a function code that describes the CEEPIPI service and that is the first parameter in the parameter list. The function code is a fullword integer (for example, 1 = init_main, 2 = call_main).

The preinitialization services offered under LE/VSE are listed in Table 53 on page 368.

An example assembler program in section "An Example Program Invocation of CEEPIPI" on page 387 illustrates invocation of CEEPIPI for the function codes `init_sub`, `call_sub`, and `term`.

Using the PIPI Table

LE/VSE uses the PIPI table to identify the routines that are candidates for execution in the preinitialized environment, as well as optionally to load the routine when it is called. It is possible to have an empty PIPI table with no entries. The PIPI table contains the names and the entry point addresses of each routine that can be executed within the preinitialized environment. Candidate routines can be present in the table when the `init_main` or `init_sub` functions are invoked, or can be added to the table using CEEPIPI(add_entry).

C Considerations

C programs that are the target of CEEPIPI(call_main) or CEEPIPI(call_sub) must be C/VSE programs (not C/370 programs).

COBOL Considerations

COBOL programs that are the target of CEEPIPI(call_main) or CEEPIPI(call_sub) must be COBOL/VSE programs (not VS COBOL II or DOS/VS COBOL programs).

PL/I Considerations

PL/I routines that are the target of CEEPIPI(call_main) or CEEPIPI(call_sub) must be PL/I VSE routines (not DOS PL/I routines).

Macros that Generate the PIPI Table

LE/VSE provides the following assembler macros to generate the PIPI table for you:

CEEXPIT
CEEXPITY
CEEXPITS

CEEXPIT: CEEXPIT generates a header for the PIPI table. This macro has no parameters.

Syntax

```
►►──table_name──CEEXPIT──────────────────────────────────────────────►◄
```

table_name
Assembler symbolic name assigned to the first word in the PIPI table. This is the value that should be used as the *ceexptbl_addr* parameter in a CEEPIPI(init_main) or a CEEPIPI(init_sub) call.

CEEXPITY: CEEXPITY generates an entry within the PIPI table.

Syntax

```
►►──CEEXPITY──(──┬──────┬──,──┬─────────────┬──)─────────────────────►◄
                 └─name─┘     └─entry_point─┘
```

name
The eight-character phase name of a routine that can be invoked within the LE/VSE preinitialized environment.

entry_point
The address of the phase that is to be invoked.

You have the option of specifying either, both, or neither of the parameters:

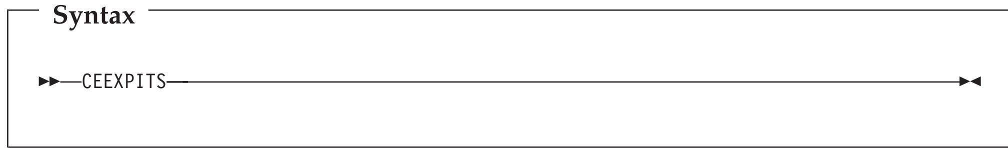
- If *name* is omitted and *entry_point* is present, the comma must be present.
- If both parameters are omitted, the entry is a candidate for assignment to the PIPI table by a call to CEEPIPI(add_entry).
- If both parameters are present, *name* is ignored and *entry_point* is used as the start of the routine.

Each invocation of the CEEXPITY macro generates a row in the PIPI table. The first entry is row 0, the second is row 1, and so on.

CEEXPITS: CEEXPITS identifies the end of the PIPI table.

This macro has no parameters.

Syntax



Reentrancy Considerations

You can make multiple calls to main routines by invoking CEEPIPI services and making multiple requests from a single PIPI table. In general, you should specify only reentrant routines for multiple invocations, or you might get unexpected results.

For instance, if you have a reentrant main routine that is invoked using CEEPIPI(call_main) and that uses external variables, then when your routine is invoked again, the external variables are re-initialized. Multiple executions of a reentrant main routine are not influenced by a previous execution of the same routine.

On the other hand, if you have a nonreentrant main routine that is invoked using CEEPIPI(call_main) and that uses external variables, then when your routine is invoked again, the external variables can potentially contain last-used values. Local variables (those contained in the object code itself) might also contain last-used values. If main routines are allowed to execute multiple times, a given execution of a routine can influence subsequent executions of the same routine.

User Exit Invocation

User exits are invoked for initialization and termination during calls to CEEPIPI as shown in Table 52.

Table 52. Invocation of User Exits during Process and Enclave Initialization and Termination

Function	When Invoked
Assembler user exit for first enclave initialization	• CEEPIPI(*init_sub*) • CEEPIPI(*init_sub_dp*) • CEEPIPI(*call_main*) • CEEPIPI(*call_sub*) or CEEPIPI(*call_sub_addr*) if a previous CEEPIPI(*call_sub*) or CEEPIPI(*call_sub_addr*) ended with stop semantics (see "Stop Semantics" on page 367)
HLL user exit	• CEEPIPI(*init_sub*) • CEEPIPI(*init_sub_dp*) • CEEPIPI(*call_main*) • CEEPIPI(*call_sub*) or CEEPIPI(*call_sub_addr*) if a previous CEEPIPI(*call_sub*) or CEEPIPI(*call_sub_addr*) ended with stop semantics
C `atexit()` functions	• CEEPIPI(*call_main*) • CEEPIPI(*call_sub*) or CEEPIPI(*call_sub_addr*), which ended stop semantics. • CEEPIPI(*term*) for environment created with CEEPIPI(*init_sub*) or CEEPIPI(*init_sub_dp*), if the last CEEPIPI(*call_sub*) or CEEPIPI(*call_sub_addr*) did not end with stop semantics

Table 52. Invocation of User Exits during Process and Enclave Initialization and Termination (continued)

Function	When Invoked
Assembler user exit for first enclave termination	• CEEPIPI(*call_main*) • CEEPIPI(*call_sub*) or CEEPIPI(*call_sub_addr*), which ended stop semantics • CEEPIPI(*term*) for environment created with CEEPIPI(*init_sub*) or CEEPIPI(*init_sub_addr*) if the last CEEPIPI(*call_sub*) or CEEPIPI(*call_sub_addr*) did not end with stop semantics
Assembler user exit for process termination	• CEEPIPI(*term*)

See Chapter 25, "Using Run-Time User Exits," on page 319 for more information about user exits.

Stop Semantics

When one of the following is issued within the preinitialized environment for subroutines:

- COBOL STOP RUN statement
- C `exit()`, `return()`, `abort()`, or signal handling function specifying a normal or abnormal termination
- PL/I STOP or EXIT

or when an unhandled condition causes termination of the (only) thread, the logical enclave is terminated. The process level of the environment is retained. LE/VSE does **not** delete those entries that were loaded explicitly by LE/VSE during the preinitialization processing.

Specifying Run-Time Options and Program Arguments

When using LE/VSE preinitialization facilities, you can specify LE/VSE run-time options in the following ways:

As installation defaults
: For information about establishing installation defaults, see "Specifying Run-time Options" on page 35.

In the assembler user exit
: For information about how to specify a list of run-time options in the assembler user exit, see "CEEBXITA Assembler User Exit Interface" on page 323.

: CEEBXITA must be linked with the routine specified in the first valid entry in the PIPI table. Any occurrences of CEEBXITA in other PIPI table entries are ignored.

As application defaults
: The CEEUOPT assembler language source program sets application defaults using the CEEXOPT macro. Like the installation defaults module, CEEDOPT, CEEUOPT can be edited and assembled to create an object module, CEEUOPT, that can be linked with an application. When the application runs, the options specified in CEEUOPT override any corresponding overridable CEEDOPT options.

CEEUOPT must be linked with the routine specified in the first valid entry in the PIPI table. Any occurrences of CEEUOPT in other PIPI table entries are ignored.

In your CEEPIPI calls
You can specify run-time options in the *runtime_opts* parameter of a CEEPIPI(*init*_sub), CEEPIPI(init_sub_dp), or CEEPIPI(*call*_main) request.

In your C source code
You can specify run-time options in `#pragma runopts` directive in your C source program. For more information, see "Specifying Run-time Options" on page 35.

In your PL/I source code
You can specify run-time options in the PLIXOPT string in your PL/I source program. For more information, see "Specifying Run-time Options" on page 35.

When using LE/VSE preinitialization facilities, you can specify program arguments in the *parm_ptr* parameter of a CEEPIPI(call_sub), CEEPIPI(call_sub_addr), or CEEPIPI(call_main) request.

Note: When using LE/VSE preinitialization facilities, LE/VSE does not honor run-time options or program arguments specified in the PARM parameter of the JCL EXEC statement.

CEEPIPI Interface

The following section describes how to invoke the CEEPIPI interface to perform the following tasks:
- Initialization
- Application invocation
- Termination
- Addition of an entry to the PIPI table

CEEPIPI preinitialization services offered under LE/VSE are listed in Table 53.

Table 53. Preinitialization Services Accessed Using CEEPIPI

Function Code	Integer Value	Service Performed
Initialization		
init_main	1	Create and initialize an environment for multiple executions of main routines
init_sub	3	Create and initialize an environment for multiple executions of subroutines
init_sub_dp	9	Create and initialize an environment for multiple executions of subroutines
Application invocation		
call_main	2	Invoke a main routine within an already initialized environment
call_sub	4	Invoke a subroutine within an already initialized environment
start_seq	7	Start a sequence of uninterruptable calls to a number of subroutines
call_sub_addr	10	Invoke a subroutine by address within an already initialized environment
Termination		

Table 53. Preinitialization Services Accessed Using CEEPIPI (continued)

Function Code	Integer Value	Service Performed
term	5	Explicitly terminate the environment without executing a user routine
end_seq	8	Terminate a sequence of uninterruptable calls to a number of subroutines
Addition of an entry to PIPI table		
add_entry	6	Dynamically add a candidate routine to execute within the preinitialized environment

Initialization

An LE/VSE environment can be initialized in two different capacities—one to allow executions of *main* routines, the other to allow multiple executions of subroutines. Each capacity is discussed below.

CEEPIPI(init_main)—Initialize for Main Routines

This invocation of CEEPIPI:

- Creates and initializes a new common run-time environment (process) that allows the execution of main routines multiple times
- Sets the environment dormant so that exceptions are percolated out of it
- Returns a token identifying the environment to the caller
- Returns a code in Register 15 indicating whether an environment was successfully initialized

Syntax

```
►►──CALL──CEEPIPI──(──init_main──,──ceexptbl_addr──,──service_rtns──,──token──)──────────►◄
```

init_main (input)
A fullword function code (integer value = 1) containing the init_main request.

ceexptbl_addr (input)
A fullword containing the address of the PIPI table to be used during initialization of the new environment. LE/VSE does not alter the user-supplied copy of the table. If an entry address is zero and the entry name is nonblank, LE/VSE searches for the routine (see “Specifying the Search Order” on page 29) and dynamically loads it. LE/VSE places the entry address in the corresponding slot of an LE/VSE-maintained table.

LE/VSE uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the entry address is zero, and the entry name is supplied, LE/VSE uses the AMODE returned by the system loader. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

service_rtns (input)
A fullword containing the address of the service routine vector or 0, if there is no service routine vector. See “Service Routines” on page 382 for more information.

token (output)
A fullword containing a unique value used to represent the environment.

The *token* parameter should be used only as input to additional calls to CEEPIPI, and should not be altered or used in any other manner.

Return Codes: Register 15 contains a return code indicating whether an environment was successfully initialized or not. Possible return codes are:

0 A new environment was successfully initialized.

4 The function code is invalid.

8 Not all addresses in the table were resolved. This can occur if a load failure was encountered or a routine within the table was generated by a non-LE/VSE-conforming HLL.

12 The version of the CEEXPIT macro used at assembly time is not supported by the version of LE/VSE that is currently running.

16 CEEPIPI was called from an active environment.

Usage Note:

- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the first valid entry in the PIPI table. Any occurrences of CEEBXITA, CEEBINT, and CEEUOPT in other PIPI table entries are ignored.

CEEPIPI(init_sub)—Initialize for Subroutines

This invocation of CEEPIPI:

- Creates and initializes a new common run-time environment (process and enclave) that allows the execution of subroutines multiple times
- Sets the environment dormant so that exceptions are percolated out of it
- Returns a token identifying the environment to the caller
- Returns a code in Register 15 indicating whether an environment was successfully initialized
- Ensures that when the environment is dormant, it is immune to other LE/VSE enclaves that are created or terminated

Syntax

```
►►—CALL—CEEPIPI—(—init_sub—,—ceexptbl_addr—,—service_rtns—,————————————►

►—runtime_opts—,—token—)————————————————————————————————————————————►◄
```

init_sub (input)
A fullword function code (integer value = 3) containing the init_sub request.

ceexptbl_addr (input)
A fullword containing the address of the PIPI table to be used during initialization of the new environment. LE/VSE does not alter the user-supplied copy of the table. If the entry point address is zero and the routine name is nonblank, LE/VSE searches for the routine (see "Specifying the Search Order" on page 29) and dynamically loads it. LE/VSE then places the entry address in the corresponding slot of an LE/VSE-maintained table.

LE/VSE uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the entry address is zero, and the

routine name is supplied, LE/VSE uses the AMODE returned by the system loader. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

service_rtns (input)
A fullword containing the address of the service routine vector. It contains 0 if there is no service routine vector. See "Service Routines" on page 382 for more information.

run-time_opts (input)
A fixed-length 255-character string containing run-time options (see *LE/VSE Programming Reference* for a list of run-time options that you can specify).

token (output)
A fullword containing a unique value used to represent the environment.

The *token* parameter should be used only as input to additional calls to CEEPIPI, and should not be altered or used in any other manner.

Return Codes: Register 15 contains a return code indicating the success or failure of the call. Possible return codes are:

0 A new environment was successfully initialized.

4 The function code is invalid.

8 Not all addresses in the table were resolved. This can occur if a load failure was encountered or a routine within the table was generated by a non-LE/VSE-conforming HLL.

12 The version of the CEEXPIT macro used at assembly time is not supported by the version of LE/VSE that is currently running.

16 CEEPIPI was called from an active environment.

Usage Note:

- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the first valid entry in the PIPI table. Any occurrences of CEEBXITA, CEEBINT, and CEEUOPT in other PIPI table entries are ignored.

CEEPIPI(init_sub_dp)—Initialize for Subroutines (Multiple Environment)

This invocation of CEEPIPI:

- Creates and initializes a new LE/VSE process and enclave to allow the execution of subroutines multiple times
- Sets the environment dormant so that exceptions are percolated out of it
- Returns a token identifying the environment to the caller
- Returns a code in Register 15 indicating whether an environment was successfully initialized
- Ensures that the environment tolerates the existence of multiple LE/VSE enclaves
- Ensures that when the environment is dormant, it is immune to other LE/VSE enclaves that are created or terminated

Multiple environments can be established only by using the CEEPIPI(init_sub_dp) as opposed to CEEPIPI(init_sub), which can establish only a single environment.

Syntax

```
►►—CALL—CEEPIPI—(—init_sub_dp—,—ceexptbl_addr—,—service_rtns—,——————►

►—runtime_opts—,—token—)——————————————————————————————————————————►◄
```

init_sub_dp (input)
A fullword function code (integer value = 9) containing the *init_sub_dp* request.

ceexptbl_addr (input)
A fullword containing the address of the PIPI table to be used during initialization of the new environment. LE/VSE does not alter the user-supplied copy of the table. If the entry point address is zero and the routine name is nonblank, LE/VSE searches for the routine (see "Specifying the Search Order" on page 29) and dynamically loads it. LE/VSE then places the entry address in the corresponding slot of an LE/VSE-maintained table.

LE/VSE uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the entry address is zero, and the routine name is supplied, LE/VSE uses the AMODE returned by the system loader. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

service_rtns (input)
A fullword containing the address of the service routine vector. It contains 0 if there is no service routine vector. See "Service Routines" on page 382 for more information.

run-time_opts (input)
A fixed-length 255-character string containing run-time options (see *LE/VSE Programming Reference* for a list of run-time options that you can specify).

token (output)
A fullword containing a unique value used to represent the environment.

The *token* parameter should be used only as input to additional calls to CEEPIPI, and should not be altered or used in any other manner.

Return Codes: Register 15 contains a return code indicating the success or failure of the call. Possible return codes are:

0 A new environment was successfully initialized.

4 The function code is invalid.

8 Not all addresses in the table were resolved. This can occur if a load failure was encountered or a routine within the table was generated by a non-LE/VSE-conforming HLL.

12 The version of the CEEXPIT macro used at assembly time is not supported by the version of LE/VSE that is currently running.

Usage Notes:

- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the first valid entry in the PIPI table. Any occurrences of CEEBXITA, CEEBINT, and CEEUOPT in other PIPI table entries are ignored.

- COBOL, PL/I, and C routines must be compiled RENT to participate in this environment.
- You can direct MSGFILE output to either SYSLST or to a unique file.
- C memory files are not shared across multiple environments.

Application Invocation

LE/VSE provides two facilities to invoke either a main routine or subroutine. When invoking main routines, the environment must have been initialized using the init_main function code. In a similar manner, when invoking subroutines, the environment must have been initialized with the init_sub function code.

CEEPIPI(call_main)—Invocation for Main Routine

This invocation of CEEPIPI invokes as a main routine the routine that you specify. The common execution environment identified by *token* is activated before the called routine is invoked, and after the called routine returns, the environment is dormant.

At termination, the currently active HLL event handlers are driven to enforce language semantics for the termination of an application such as closing files and freeing storage. The process level is made dormant rather than terminated. The thread and enclave levels are terminated. The assembler user exit is driven with the function code for first enclave termination. (See Chapter 25, “Using Run-Time User Exits,” on page 319 for more information about user exits.)

Syntax

```
►►──CALL──CEEPIPI──(──call_main──,──ceexptbl_index──,──token──,──────────────►
►──runtime_opts──,──parm_ptr──,──enclave_return_code──,──────────────────────►
►──enclave_reason_code──,──appl_feedback_code──)─────────────────────────────►◄
```

call_main (input)
A fullword function code (integer value = 2) containing the call_main request.

ceexptbl_index (input)
A fullword containing the row number within the PIPI table of the entry that should be invoked. The index starts at 0.

Note that each invocation of the CEEXPITY macro generates a row in the PIPI table. The first entry is row 0, the second is row 1 and so on. A call to CEEPIPI(add_entry) to add an entry to the PIPI table also returns a row number in the *ceexptbl_index* parameter.

token (input)
A fullword with the value of the token returned by CEEPIPI(init_main).

The *token* parameter must identify a previously preinitialized environment that is not active at the time of the call.

runtime_opts (input)
A fixed-length 255-character string containing run-time options. (See *LE/VSE Programming Reference* for a list of run-time options that you can specify.)

parm_ptr (input)
A fullword parameter list pointer or 0 (zero) that is placed in Register 1 when the main routine is executed.

The parameter list that is passed must be in a format that HLL subroutines expect (for example, in an `argc, argv` format for C routines).

enclave_return_code (output)
A fullword containing the enclave return code returned by the called routine when it finished executing. For more information about return codes, see "Managing Return Codes in LE/VSE" on page 68.

enclave_reason_code (output)
A fullword containing the enclave reason code returned by the environment when the routine finished executing. For more information about reason codes, see "Managing Return Codes in LE/VSE" on page 68.

appl_feedback_code (output)
A 96-bit condition token indicating why the application terminated.

Return Codes: A return code is provided in Register 15 and can contain the following values:

0 The environment was activated and the routine called.

4 The function code is invalid.

8 CEEPIPI was called from an LE/VSE-conforming HLL.

12 The indicated environment was initialized for subroutines. No routine was executed.

16 The *token* parameter is invalid.

20 The index points to an entry that is invalid or empty.

24 The index that was passed is outside the range of the table.

Usage Notes:

- The NOEXECOPS and CBLOPTS run-time options (see *LE/VSE Programming Reference*) are ignored since the parameter inbound to the application and the run-time options are separated already. Therefore, NOEXECOPS and CBLOPTS do not affect the parameter string format. See "C PLIST and EXECOPS Interactions" on page 405 for more information.
- For more information about return codes, see "Managing Return Codes in LE/VSE" on page 68.

CEEPIPI(call_sub)—Invocation for Subroutines

This invocation of CEEPIPI invokes as a subroutine the routine that you specify. The common run-time environment identified by *token* is activated before the called routine is invoked, and after the called routine returns, the environment is dormant.

The enclave is terminated when an unhandled condition is encountered or a STOP statement is executed. (See "Stop Semantics" on page 367 for more information.) However, the process level is maintained. The next call to CEEPIPI(call_sub) initializes a new enclave.

Syntax

```
►►—CALL—CEEPIPI—(—call_sub—,—ceexptbl_index—,—token—,—parm_ptr—,————————►

►—sub_ret_code—,—sub_reason_code—,—sub_feedback_code—)————————————————►◄
```

call_sub (input)
A fullword function code (integer value = 4) containing the call_sub request for a subroutine.

ceexptbl_index (input)
A fullword containing the row number of the entry within the PIPI table that should be invoked. The index starts at 0.

token (input)
A fullword with the value of the token returned when the common run-time environment is initialized. This token is initialized by the CEEPIPI(init_sub) or CEEPIPI(init_sub_dp).

The *token* parameter must identify a previously preinitialized environment that is not active at the time of the call. You must not alter the value of the token.

parm_ptr (input)
A parameter list pointer or 0 (zero) that is placed in register 1 when the routine is executed.

C users need to follow the subroutine linkage convention for C—assembler ILC applications as outlined in *LE/VSE C Run-Time Programming Guide*.

sub_ret_code (output)
The subroutine return code.

If the enclave is terminated due to an unhandled condition or a STOP statement, this contains the enclave return code for termination.

sub_reason_code (output)
The subroutine reason code. This is 0 for normal subroutine returns. If the enclave is terminated due to an unhandled condition or a STOP statement, this contains the enclave reason code for termination.

sub_feedback_code (output)
The feedback code for enclave termination. This is the CEE000 feedback code for normal subroutine returns. If the enclave is terminated due to an unhandled condition or a STOP statement, this contains the enclave feedback code for termination.

A return code is provided in Register 15 and can contain the following values:

0 The environment was activated and the routine called.

4 The function code is invalid.

8 CEEPIPI was called from an LE/VSE-conforming HLL.

12 The indicated environment was initialized for main routines. No routine was executed.

16 The *token* parameter is invalid.

20 The index points to an entry that is invalid or empty.

24 The index passed is outside the range of the table.

28 The enclave was terminated but the process level persists.

This value indicates the enclave was terminated while the process was retained. This can occur due to a STOP statement being issued or due to an unhandled condition. The *sub_ret_code*, *sub_reason_code*, and *sub_feedback_code* indicate this action.

Usage Notes:

- The enclave terminates if the subroutine issues a STOP statement or if there is an unhandled condition. However, the process level is not terminated. When the enclave level is terminated, any subsequent invocation creates a new enclave by using the same run-time options used in the creation of the first enclave. LE/VSE does not delete any user routines that were loaded into the PIPI table. (See “Stop Semantics” on page 367.)
- External data must be in its original state before preinitialization.

CEEPIPI(call_sub_addr)—Invocation for Subroutines by Address

This invocation of CEEPIPI invokes as a subroutine the routine that you specify. The common run-time environment identified by *token* is activated before the called routine is invoked, and after the called routine returns, the environment is dormant.

The enclave is terminated when an unhandled condition is encountered or a STOP statement is executed. (See “Stop Semantics” on page 367 for more information.) However, the process level is maintained; only the enclave level terminates.

Syntax

```
►►—CALL—CEEPIPI—(—call_sub_addr—,—routine_addr—,—token—,—parm_ptr—,——————►

►—sub_ret_code—,—sub_reason_code—,—sub_feedback_code—)——————————————————►◄
```

call_sub_addr (input)
A fullword function code (integer value = 10) containing the call_sub request for a subroutine.

routine_addr (input)
A doubleword containing the address of the routine that should be invoked. The first fullword contains the entry point address and the second fullword must be zero.

token (input)
A fullword with the value of the token returned by CEEPIPI(init_sub) or CEEPIPI(init_sub_dp) when the common run-time environment is initialized.

The *token* parameter must identify a previously preinitialized environment that is not active at the time of the call. You must not alter the value of the token.

parm_ptr (input)
A parameter list pointer or 0 (zero) that is placed in register 1 when the routine is executed.

C users are advised to follow the subroutine linkage convention for C—assembler ILC applications as outlined in *LE/VSE C Run-Time Programming Guide*.

sub_ret_code (output)
The subroutine return code.

If the enclave is terminated due to an unhandled condition or a STOP statement, this contains the enclave return code for termination.

sub_reason_code (output)
The subroutine reason code. This is 0 for normal subroutine returns. If the enclave is terminated due to an unhandled condition or a STOP statement, this contains the enclave reason code for termination.

sub_feedback_code (output)
The feedback code for enclave termination. This is the CEE000 feedback code for normal subroutine returns. If the enclave is terminated due to an unhandled condition or a STOP statement, this contains the enclave feedback code for termination.

Return Codes: A return code is provided in register 15 and can contain the following values:

0 The environment was activated and the routine called.

4 The function code is invalid.

8 CEEPIPI was called from an LE/VSE-conforming HLL.

12 The indicated environment was initialized for main routines. No routine was executed.

16 The *token* parameter is invalid.

28 The enclave was terminated but the process level persists.

This value indicates the enclave was terminated while the process was retained. This can occur due to a STOP statement being issued or due to an unhandled condition. The *sub_ret_code, sub_reason_code,* and *sub_feedback_code* indicate this action.

Usage Notes:

- The enclave terminates if the subroutine issues a STOP statement or if there is an unhandled condition. However, the process level is not terminated. When the enclave level is terminated, any subsequent invocation creates a new enclave using the same run-time options used in the creation of the first enclave. LE/VSE does not delete any user routines that were loaded into the PIPI table. (See “Stop Semantics” on page 367.)
- External data must be in its original state before preinitialization.

CEEPIPI(start_seq)—Start a Sequence of Calls

This invocation of CEEPIPI declares that a sequence of uninterrupted calls is made to a number of subroutines by this driver program to the same preinitialized environment. This minimizes the overhead between calls by performing as much activity as possible at the start of a sequence of calls.

Syntax

```
►►—CALL—CEEPIPI—(—start_seq—,—token—)————————————————————————►◄
```

start_seq (input)
A fullword function code (integer value = 7) containing the start_seq request.

token (input)
A fullword with the value of the token returned by CEEPIPI(init_sub_dp) when the common run-time environment is initialized.

The *token* parameter must identify a previously preinitialized environment for subroutines that is dormant at the time of the call.

Return Codes: A return code is provided in register 15 and can contain the following values:

0 The environment was prepared for a sequence of calls.

4 The function code is invalid.

8 The indicated environment was already active. No action taken.

16 The *token* parameter is invalid.

20 Sequence already started using *token*.

Usage Notes:

- CEEPIPI(start_seq) can be used only in conjunction with an LE/VSE environment initialized by CEEPIPI(init_sub_dp) function code. A return code 4 is set for environments not initialized by CEEPIPI(init_sub_dp).
- CEEPIPI(start_seq) minimizes the overhead between calls by allowing LE/VSE to perform as much activity as possible at the start of the sequence of calls.
- Only CEEPIPI(call_sub) or CEEPIPI(call_sub_addr) invocations are allowed between the CEEPIPI(start_seq) and CEEPIPI(end_seq) calls.
- The same *token* must be passed for all invocations of CEEPIPI(call_sub) or CEEPIPI(call_sub_addr) between the CEEPIPI(start_seq) and CEEPIPI(end_seq) function codes. You can vary the routine invoked.

CEEPIPI(end_seq)—End a Sequence of Calls

This invocation of CEEPIPI declares that a sequence of uninterrupted calls to subroutines by this driver program has finished.

Syntax

```
►►──Call──CEEPIPI──(──end_seq──,──token──)─────────────────────────────────►◄
```

end_seq (input)
A fullword function code (integer value = 8) containing the end_seq request

token (input)
A fullword with the value of the token returned by CEEPIPI(init_sub_dp) when the common run-time environment is initialized.

The *token* parameter must identify a previously preinitialized environment that was prepared for multiple calls via the CEEPIPI(start_seq) call.

Return Codes: A return code is provided in register 15 and can contain the following values:

0 The environment is no longer prepared for a sequence of calls.

4 The function code is invalid.

8 The indicated environment was already active. No action taken.

16 The *token* parameter is invalid.

20 The specified *token* was not used in a start_seq call.

Usage Notes:

- CEEPIPI(end_seq) can be used only in conjunction with an LE/VSE environment initialized by an CEEPIPI(init_sub_dp) function code. A return code of 4 is set for environments initialized by other than CEEPIPI(init_sub_dp).
- Only CEEPIPI(call_sub) or CEEPIPI(call_sub_addr) invocations are allowed between the CEEPIPI(start_seq) and CEEPIPI(end_seq) calls.
- This function can be called from an active environment if the PIPI environment indicated by *token* was created with the CEEPIPI(init_sub_dp) function.

Termination

An LE/VSE environment can be terminated by calling CEEPIPI with a termination request.

CEEPIPI(term)—Terminate Environment

This invocation of CEEPIPI terminates the environment identified by the value given in *token*. This service is used for terminating environments created for subroutines or main routines.

Syntax

```
►►──CALL──CEEPIPI──(──term──,──token──,──env_return_code──)──────────────────────►◄
```

term (input)
A fullword function code (integer value = 5) containing the termination request.

token (input)
A fullword with the value of the token of the environment to be terminated. This token is returned by a CEEPIPI(init_main), CEEPIPI(init_sub), or CEEPIPI(init_sub_dp) request during the initialization call.

The *token* parameter must identify a previously preinitialized environment that is dormant at the time of the call.

env_return_code (output)
A fullword integer which is set to the return code from the environment termination.

If the environment was initialized for a main routine or a subroutine, and the last CEEPIPI(call_sub) or CEEPIPI(call_sub_addr) issued stop semantics, the value of *env_return_code* is zero.

If the environment was initialized for a subroutine, and the last CEEPIPI(call_sub) or CEEPIPI(call_sub_addr) did not terminate with stop semantics, *env_return_code* contains the same value as that in *sub_ret_code* from the last CEEPIPI(call_sub) or CEEPIPI(call_sub_addr).

Upon return, Register 15 contains a return code indicating the success or failure of this request and can contain the following values:

0 The environment was activated and termination was requested.

4 Invalid function code.

8 CEEPIPI was called from an LE/VSE-conforming routine.

16 The *token* parameter is invalid.

Usage Notes:

- All resources obtained are released when the environment terminates.
- All routines loaded by LE/VSE are deleted when the environment terminates.
- Subsequent references to *token* by preinitialization services result in an error indicating the token is invalid.

Adding an Entry to the PIPI Table

You can add an entry to the PIPI table by calling CEEPIPI with an add_entry request.

CEEPIPI(add_entry)—Add an Entry to the PIPI Table

This invocation of CEEPIPI adds an entry for the environment represented by *token* in the LE/VSE-maintained table. If a routine entry address is not provided, the routine name is used to dynamically load the routine and add it to the PIPI table. The PIPI table index for the new entry is returned to the calling routine.

Syntax

```
►►—CALL—CEEPIPI—(—add_entry—,—token—,—routine_name—,——————————————►

►—routine_entry—,—ceexptbl_index—)—————————————————————————————————►◄
```

add_entry (input)
A fullword function code (integer value = 6) containing the add_entry request.

token (input)
A fullword with the value of the token associated with the environment that adds this new routine. This token is returned by a CEEPIPI(init_main), CEEPIPI(init_sub), or CEEPIPI(init_sub_dp) request.

The *token* parameter must identify a previously preinitialized environment that is dormant at the time of the call.

routine_name (input)
A character string of length 8, left-justified and padded right with blanks, containing the name of the routine. To indicate the absence of the name, this field should be blank. If *routine_entry* is zero, this is used as the load name.

routine_entry (input/output)
The routine entry address that is added to the PIPI table. If *routine_entry* is zero on input, *routine_name* is used as the load name. On output, *routine_entry* is set to the load address of *routine_name*.

ceexptbl_index (output)
The index to the PIPI table where this routine was added. If the return code is nonzero, this value is indeterminate. The index starts at zero.

LE/VSE uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the *routine_entry* is zero, and the *routine_name* is supplied LE/VSE uses the AMODE returned by the system loader. If the *routine_entry* is supplied, you must provide the AMODE in the high-order bit of the address.

Return Codes: Upon return, Register 15 contains a return code indicating the success or failure of this request and can contain one of the following values:

0 The routine was added to the PIPI table.

4 Invalid function code.

8 CEEPIPI was called from an LE/VSE-conforming routine.

12 The routine did not contain a common run-time environment PPA style prolog. The PIPI table was not updated. *routine_entry* is set to the address of the loaded routine.

16 The *token* parameter is invalid.

20 The *routine_name* contains only blanks and the *routine_entry* was zero. The PIPI table was not updated.

24 The *routine_name* was not found or there was a load failure. The PIPI table was not updated.

28 The PIPI table is full. No routine was added to the table, nor was any routine loaded by LE/VSE.

Usage Notes:

- The PIPI table is built using the macros described in this chapter. Therefore, its size is under the control of your application, not LE/VSE.
- None of the routines in the PIPI table can be nested routines. All routines must be external routines.

Deleting an Entry from the PIPI Table

You can delete an entry from the PIPI table by calling CEEPIPI with a delete_entry request. The entry is then available for subsequent CEEPIPI(add_entry) functions.

CEEPIPI(delete_entry)—Delete an Entry from the PIPI Table

This invocation of CEEPIPI removes an entry for the environment represented by *token* in the LE/VSE-maintained table.

Syntax

```
►►──CALL──CEEPIPI──(──delete_entry──,──token──,──ceexptbl_index──)──────────────────►◄
```

delete_entry (input)
A fullword function code (integer value = 11) containing the delete_entry request.

token (input)
A fullword with the value of the token of the environment. This is the token returned by a CEEPIPI(init_sub) or CEEPIPI(init_sub_dp) request.

ceexptbl_index (output)
The index into the PIPI table of the entry to delete.

Return Codes: Upon return, Register 15 contains a return code indicating the success or failure of this request and can contain one of the following values:

0 The routine was deleted from the PIPI table.

4 Invalid function code.

8 CEEPIPI was called from an active environment. No entries were deleted from the PIPI table.

16 The *token* parameter is invalid.

20 The PIPI table entry indicated by *ceexptbl_index* was empty.

24 The index passed is outside the range of the table.

28 The system request to delete the routine failed; the routine was not deleted from the PIPI table.

Usage Notes:

- The token must identify a previously-preinitialized environment that is dormant at the time of the call.
- If the routine indicated by *ceexptbl_index* had been loaded by CEEPIPI, it will be deleted.
- Only C or PL/I reentrant fetchable subroutines are support targets of a delete_entry request. COBOL target load modules are not supported.

Service Routines

Under LE/VSE, you can specify several service routines to execute a main routine or subroutine in the preinitialized environment. To use the routines, specify a list of addresses of the routines in a service routine vector as shown in Figure 123.

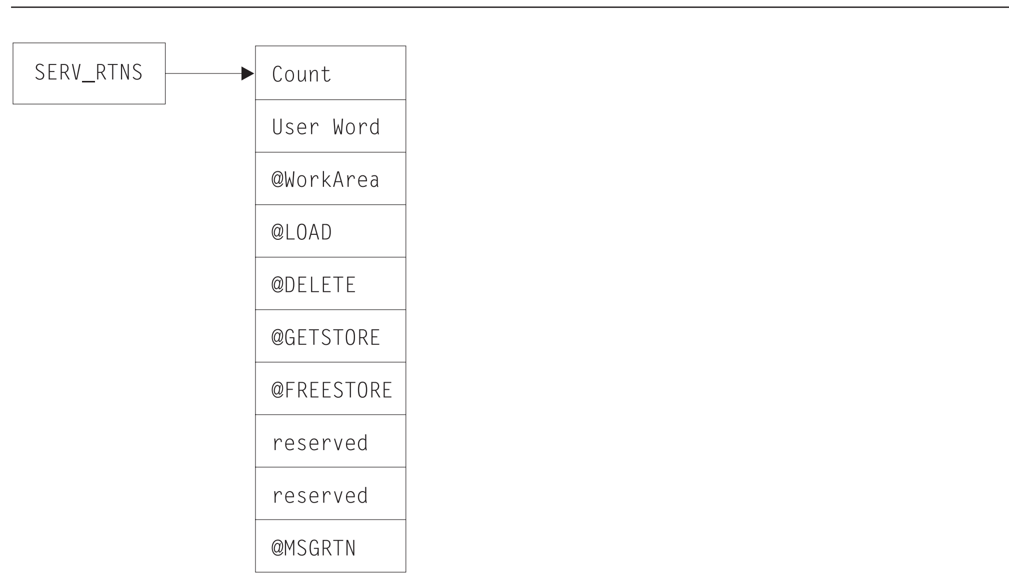


Figure 123. Format of Service Routine Vector

The service routine vector is composed of a list of fullword addresses of routines that are used instead of LE/VSE service routines. The list of addresses is preceded by the number of the addresses in the list, as specified in the *count* field of the vector. The *service_rtns* parameter that you specify in calls to CEEPIPI(init_main) and CEEPIPI(init_sub) contains the address of the vector itself. If this pointer is specified as zero (0), LE/VSE routines are used instead of the service routines shown in Figure 123.

The @GETSTORE and @FREESTORE service routines must be specified together; if one is zero, the other is automatically ignored. The same is true for the @LOAD

and @DELETE service routines. You should be aware that if you replace the program management routines, these routines might not account for all the storage obtained for use with the application. Program management obtains virtual storage for the load module. This storage will not be managed by the user-replaced storage management routines.

The service routines must be AMODE(ANY) / RMODE(24).

Count
: A fullword binary number representing the number of fullwords that follow. The *count* does not include itself. In Figure 123 on page 382, the count is 9. For each vector slot, a zero represents the absence of the routine, a nonzero represents the presence of a routine.

User Word
: A fullword that is passed to the service routines. The *user word* is provided as a means for your routine to communicate to the service routines.

@WorkArea
: An address of a work area of at least 256 bytes that is doubleword aligned. The first word of the area contains the length of the area provided. This parameter is required if service routines are present in the service routine vector.

@LOAD
: This routine loads named routines for program management. Each time the @LOAD routine is called, the parameter list that is passed to the routine contains the following:

  Name_addr
  : The fullword address of the name of the phase to load (input).

  Name_length
  : A fixed binary(31) length of the phase name (input).

  User_word
  : A fullword user field (input).

  Rsvd_word
  : A fullword reserved for future use (input). This must be specified as zero (0).

  Entry_point
  : The fullword entry point address of the loaded routine (output).

  Module_size
  : The fixed binary(31) size of the phase that was loaded (output).

  Return code
  : The fullword return code from the @LOAD service (output).

  Reason code
  : The fullword reason code from the @LOAD service (output).

  The return and reason codes set by the @LOAD service routine supplied with LE/VSE are listed in Table 54.

Table 54. Return and Reason Codes

Return Code	Reason Code[1]	Description
0	0	Successful
8	20	Unsuccessful—phase not found
12	4	Unsuccessful—the size of the (real) partition GETVIS is 0KB
12	8	Unsuccessful—the length of the phase exceeds the GETVIS area

Table 54. Return and Reason Codes (continued)

Return Code	Reason Code[1]	Description
12	12	Unsuccessful—insufficient storage in the GETVIS area
12	16	Unsuccessful—the partition CDLOAD directory is full and there is no system GETVIS available to allocate a new directory
16	24	Unsuccessful—load request failed trying to load a move-type phase

Note:

1. The reason codes are the return codes received from the CDLOAD system macro.

@DELETE

This routine deletes named routines for program management. Each time the @DELETE routine is called, the parameter list that is passed to the routine contains the following:

Name_addr
The fullword address of the phase name to be deleted (input).

Name_length
A fixed binary(31) length of phase name (input).

User_word
A fullword user field (input).

Rsvd_word
A fullword reserved for future use (input). Must be zero.

Return code
The return code from the @DELETE service (output).

Reason code
The fullword reason code from the @DELETE service (output).

The return and reason codes set by the @DELETE service routine supplied with LE/VSE are listed in Table 55.

Table 55. Return and Reason Codes

Return Code	Reason Code	Description
0	0	Successful[2]
4	4[1]	Unsuccessful—maximum load count for phase exceeded
4	8	Unsuccessful—insufficient program storage available to determine if phase is in the SVA
4	20[1]	Unsuccessful—phase not found

Note:

1. This reason code is the return code received from the CDDELETE system macro.
2. This return code and reason code are also set if the phase is found in the SVA.

@GETSTORE

This routine allocates storage on behalf of the storage manager. This routine can rely on the caller to provide a save area, which can be the **@WorkArea**. Each time the @GETSTORE routine is called, the parameter list that is passed to the routine contains the following:

Amount
A fixed binary(31) amount of storage requested (input).

If the specified value is less than zero, this is a request for the maximum amount of available contiguous storage. The absolute value of **Amount** is the minimum amount of storage required.

Rsvd_word
A fullword reserved for future use (input).

User_word
A fullword user field (input).

Flags A fullword flag area (input).

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

Stg_address
The fullword address of the storage obtained or zero (output).

Obtained
A fixed binary(31) number of bytes obtained (output).

Return code
The fullword return code from the @GETSTORE service (output).

Reason code
The fullword reason code from the @GETSTORE service (output).

The return and reason codes set by the @GETSTORE service routine supplied with LE/VSE are listed in Table 56.

Table 56. Return and Reason Codes

Return Code	Reason Code	Description
0	0	Successful
16	4	Unsuccessful—the size of the (real) partition GETVIS is 0KB
16	8	Unsuccessful—the specified length exceeds the GETVIS area
16	12	Unsuccessful—no more virtual storage is available in the GETVIS area

@FREESTORE
This routine frees storage on behalf of the storage manager. Each time the @FREESTORE routine is called, the parameter list that is passed to the routine contains the following:

Amount
The fixed binary(31) amount of storage to free (input).

Rsvd_word
A fullword reserved for future use (input).

User_word
A fullword user field (input).

Stg_address
The fullword address of the storage to free (input).

Return code
The fullword return code from the @FREESTORE service (output).

Reason code
The fullword reason code from the @FREESTORE service (output).

The return and reason codes set by the @FREESTORE service routine supplied with LE/VSE are listed in Table 57 on page 386.

Table 57. Return and Reason Codes

Return Code	Reason Code	Description
0	0	Successful
16	4	Unsuccessful—the size of the (real) partition GETVIS is 0KB
16	8	Unsuccessful—the value specified in **Amount** is less than zero
16	12	Unsuccessful—the specified address is not within the GETVIS area or the address is not a multiple of 128
16	16	Unsuccessful—the specified storage block to be released (**Stg_address** + **Amount**) exceeds the GETVIS area

@MSGRTN
This routine allows error messages to be processed by the caller of the application. Each time the @MSGRTN routine is called, the parameter list that is passed to the routine contains the following:

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

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

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

User_word
A fullword user field (input).

Line_length
The fixed binary(31) size of the output line length. This is used when the **Message** pointer is zero (output).

Return code
The fullword return code from the @MSGRTN service (output).

Return code
The fullword reason code from the @MSGRTN service (output).

The return and reason codes set by the @MSGRTN service routine supplied with LE/VSE are listed in Table 58.

Table 58. Return and Reason Codes

Return Code	Reason Code	Description
0	0	Successful
16	4	Unsuccessful—uncorrectable error occurred

An Example Program Invocation of CEEPIPI

In the following example, the assembler program ASMPIPI ASSEMBLE invokes CEEPIPI to:
- Initialize a subroutine environment under LE/VSE
- Load and call a reentrant HLL subroutine
- Terminate the LE/VSE environment

Following the assembler program are examples of the program HLLPIPI written in C, COBOL, and PL/I. You can use the assembler program to call the HLL versions of HLLPIPI.

```
*COMPILATION UNIT:  LEASMPIP
**********************************************************************
*                                                                    *
*   Function :  CEEPIPI - Initialize the PIPI environment,           *
*                         call a PIPI HLL program, and terminate     *
*                         the environment.                           *
*                                                                    *
* 1.Call CEEPIPI to initialize a subroutine environment under LE.    *
* 2.Call CEEPIPI to load and call a reentrant HLL subroutine.        *
* 3.Call CEEPIPI to terminate the LE PIPI environment.               *
*                                                                    *
* Note:  ASMPIPI is not reentrant.                                   *
*                                                                    *
**********************************************************************
*
* ====================================================================
* Standard program entry conventions.
* ====================================================================
ASMPIPI  CSECT
         STM   R14,R12,12(R13)    Save caller's registers
         LR    R12,R15            Get base address
         USING ASMPIPI,R12        Identify base register
         ST    R13,SAVE+4         Back-chain the save area
         LA    R15,SAVE           Get addr of this routine's save area
         ST    R15,8(R13)         Forward-chain in caller's save area
         LR    R13,R15            R13 -> save area of this routine
*
* Load LE CEEPIPI service routine into main storage.
*
         CDLOAD CEEPIPI           Load CEEPIPI routine dynamically
         ST    R0,PPRTNPTR        Save the addr of CEEPIPI routine
*
* Initialize an LE PIPI subroutine environment.
*
INIT_ENV EQU   *
         LA    R5,PPTBL           Get address of PIPI Table
         ST    R5,@CEXPTBL        Ceexptbl-addr -> PIPI Table
         L     R15,PPRTNPTR       Get address of CEEPIPI routine
*                                 Invoke CEEPIPI routine
         CALL  (15),(INITSUB,@CEXPTBL,@SRVRTNS,RUNTMOPT,TOKEN)
*                                 Check return code:
         LTR   R2,R15               Is R15 = zero?
         BZ    CSUB                 Yes (success).. go to next section
*                                   No (failure).. issue message
         WTO   'ASMPIPI : call to CEEPIPI(INIT_SUB) failed',ROUTCDE=2
         C     R2,=F'8'           Check for partial initialization
         BE    TSUB                 Yes.. go do PIPI termination
*                                   No.. issue message & quit
         WTO   'ASMPIPI : INIT_SUB failure RC is not 8.',ROUTCDE=2
         DUMP  RC=(2)             Cancel with bad RC and dump memory
```

Figure 124. Assembler Driver that Creates a Preinitialized Environment (Part 1 of 3)

```
*
* Call the subroutine, which is loaded by LE
*
CSUB     EQU   *
         L     R15,PPRTNPTR       Get address of CEEPIPI routine
         CALL  (15),(CALLSUB,PTBINDEX,TOKEN,PARMPTR,                   X
               SUBRETC,SUBRSNC,SUBFBC)   Invoke CEEPIPI routine
*                                 Check return code:
         LTR   R2,R15               Is R15 = zero?
         BZ    TSUB                 Yes (success).. go to next section
*                                   No (failure).. issue message & quit
         WTO   'ASMPIPI : call to CEEPIPI(CALL_SUB) failed',ROUTCDE=2
         DUMP  RC=(2)             Cancel with bad RC and dump memory
*
* Terminate the environment.
*
TSUB     EQU   *
         L     R15,PPRTNPTR       Get address of CEEPIPI routine
         CALL  (15),(TERM,TOKEN,ENV_RC)  Invoke CEEPIPI routine
*                                 Check return code:
         LTR   R2,R15               Is R15 = zero ?
         BZ    DONE                 Yes (success).. go to next section
*                                   No (failure).. issue message & quit
         WTO   'ASMPIPI : call to CEEPIPI(TERM) failed',ROUTCDE=2
         DUMP  RC=(2)             Cancel with bad RC and dump memory
*
* Standard exit code.
*
DONE     EQU   *
         LA    R15,0                Passed return code for system
         L     R13,SAVE+4           Get address of caller's save area
         L     R14,12(R13)          Reload caller's register 14
         LM    R0,R12,20(R13)       Reload caller's registers 0-12
         BR    R14                  Branch back to caller
*
* ===================================================================
* CONSTANTS and SAVE AREA.
* ===================================================================
SAVE     DC    18F'0'
PPRTNPTR DS    A                  Save the address of CEEPIPI routine
*
* Parameters passed to a CEEPIPI(INIT_SUB) call.
*
INITSUB  DC    F'3'               Function code to initialize for subr
@CEXPTBL DC    A(PPTBL)           Address of PIPI Table
@SRVRTNS DC    A(0)               Addr of service-rtns vector, 0 = none
RUNTMOPT DC    CL255' '           Fixed length string of runtime optns
TOKEN    DS    F                  Unique value returned (output)
*
* Parameters passed to a CEEPIPI(CALL_SUB) call.
*
CALLSUB  DC    F'4'               Function code to call subroutine
PTBINDEX DC    F'0'               The row number of PIPI Table entry
PARMPTR  DC    A(0)               Pointer to @PARMLIST or zero if none
SUBRETC  DS    F                  Subroutine return code (output)
SUBRSNC  DS    F                  Subroutine reason code (output)
SUBFBC   DS    3F                 Subroutine feedback token (output)
*
* Parameters passed to a CEEPIPI(TERM) call.
*
TERM     DC    F'5'               Function code to terminate
ENV_RC   DS    F                  Environment return code (output)
*
```

Figure 124. Assembler Driver that Creates a Preinitialized Environment (Part 2 of 3)

```
* ===================================================================
* PIPI Table.
* ===================================================================
PPTBL    CEEXPIT  ,               PIPI Table with index
         CEEXPITY HLLPIPI,0       0 = dynamically loaded routine with
*                                     re-entrant option
         CEEXPITS ,               End of PIPI table
*
*
         LTORG
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END   ASMPIPI
```

Figure 124. Assembler Driver that Creates a Preinitialized Environment (Part 3 of 3)

HLLPIPI Examples

```
/*Module/File Name:  EDCPIPI  */
 /*********************************************************/
 /*                                                       */
 /* HLLPIPI is called by an assembler program, ASMPIPI.   */
 /* ASMPIPI uses the LE preinitialized program            */
 /* subroutine call interface. HLLPIPI can be written     */
 /* in COBOL, C, or PL/I.                                 */
 /*                                                       */
 /*********************************************************/
#include <stdio.h>
#include <string.h>
#include <time.h>
HLLPIPI ()
{
 printf ( "C subroutine beginning\n" );
 printf ( "Called using LE PIPI call\n" );
 printf ( "Subroutine interface.\n" );
 printf ( "C subroutine returns to caller\n" );
}
```

Figure 125. C Subroutine Called by ASMPIPI

```
CBL LIB,APOST
      *Module/File Name: IGZTPIPI
      ***********************************************************
      *                                                         *
      * HLLPIPI is called by an assembler program, ASMPIPI.     *
      * ASMPIPI uses the LE preinitialized program              *
      * subroutine call interface. HLLPIPI can be written       *
      * in COBOL, C, or PL/I.                                   *
      *                                                         *
      ***********************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HLLPIPI.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       PROCEDURE DIVISION.
           DISPLAY 'COBOL subroutine beginning'.
           DISPLAY 'Called using LE PIPI '.
           DISPLAY 'Call subroutine interface.'.
           DISPLAY 'COBOL program returns to caller.'.

           GOBACK.
```

Figure 126. COBOL Program Called by ASMPIPI

```
/*Module/File Name:  IBMPIPI                              */
/*********************************************************/
/*                                                       */
/* HLLPIPI is called by an assembler program, ASMPIPI.    */
/* ASMPIPI uses the LE preinitialized program            */
/* subroutine call interface. HLLPIPI can be written     */
/* in COBOL, C, or PL/I.                                 */
/*                                                       */
/*********************************************************/
 HLLPIPI: PROC OPTIONS(FETCHABLE);
          DCL RESULT FIXED BIN(31,0) INIT(0);
          PUT SKIP LIST
              ('HLLPIPI  : PLI subroutine beginning.');
          PUT SKIP LIST
              ('HLLPIPI  : Called LE PIPI Call ');
          PUT SKIP LIST
              ('HLLPIPI  : Subroutine interface.   ');
          PUT SKIP LIST
              ('HLLPIPI  : PLI program returns to caller.');
          RETURN;
 END HLLPIPI;
```

Figure 127. PL/I Routine Called by ASMPIPI

Chapter 28. Using Nested Enclaves

An enclave is a logical run-time structure that supports the execution of a collection of routines (see Chapter 9, "Program Management Model," on page 75 for a detailed description of LE/VSE enclaves).

LE/VSE Release 4 explicitly supports the execution of a single enclave within an LE/VSE process. However, by using the system services and language constructs described in this chapter, you can create an additional, or nested, enclave and initiate its execution within the same process.

The enclave that issues a call to system services or language constructs to create a nested enclave is called the *parent* enclave. The nested enclave that is created is called the *child* enclave. The child must be a main routine; a link to a subroutine by commands and language constructs is not supported under LE/VSE.

Understanding the Basics

In LE/VSE, you can use the following methods to create a child enclave:

- Under CICS, the EXEC CICS LINK and EXEC CICS XCTL commands (see *CICS Transaction Server for VSE/ESA Application Programming Guide* for more information about these commands)
- In the batch environment, the C `system()` function (see *LE/VSE C Run-Time Library Reference* and *LE/VSE C Run-Time Programming Guide* for more information about `system()`)

If the target routine of any of these commands is not written in an LE/VSE-conforming HLL or LE/VSE-conforming assembler, no nested enclave is created.

COBOL Considerations

In the batch environment, DOS/VS COBOL routines are supported in a single enclave only.

Determining the Behavior of Child Enclaves

If you want to create a child enclave, you need to consider the following factors:

- The language of the main routine in the child enclave
- The sources from which each type of child enclave gets run-time options
- The default condition handling behavior of each type of child enclave
- The setting of the TRAP run-time option in the parent and the child enclave

All of these interrelated factors affect the behavior, particularly the condition handling, of the created enclave. The sections that follow describe how the child enclaves created by each method (EXEC CICS LINK, EXEC CICS XCTL, and C `system()` function) will behave.

© Copyright IBM Corp. 1991, 2005

Creating Child Enclaves Using EXEC CICS LINK or EXEC CICS XCTL

If your C, COBOL, or PL/I application uses EXEC CICS commands, you must also link-edit the EXEC CICS interface stub, DFHELII, with your application. To be link-edited with your application, DFHELII must be available in the object sublibrary search chain.

See *CICS Transaction Server for VSE/ESA Application Programming Guide* for more information about the EXEC CICS LINK and EXEC CICS XCTL commands.

How Run-Time Options Affect Child Enclaves

The child enclave gets its run-time options from one of the sources discussed in "Specifying Run-Time Options under CICS" on page 300. The run-time options are completely independent of the creating enclave, and can be set on an enclave-by-enclave basis.

Some of the methods for setting run-time options might slow down your transaction. Follow these suggestions to improve performance:

- If you need to specify options in CEEUOPT, specify only those options that are different from system defaults.
- Before putting transactions into production, request a storage report (using the RPTSTG run-time option) to minimize the number of GETMAINs and FREEMAINs required by the transactions.
- Ensure that VS COBOL II transactions are not link-edited with IGZETUN, which is no longer supported and which causes an informational message to be logged. Logging this message for every transaction inhibits system performance.

How Conditions Arising in Child Enclaves Are Handled

This section describes the default condition handling for child enclaves created by EXEC CICS LINK or EXEC CICS XCTL.

Condition handling varies depending on the source of the condition, and whether or not an EXEC CICS HANDLE ABEND is active:

- If a software condition of severity 2 or greater occurs, LE/VSE condition handling takes place. If the condition remains unhandled, the problem is not percolated to the parent enclave. The CICS thread is terminated with an abend. These actions take place even if a CICS HANDLE ABEND is active, because CICS HANDLE ABEND does not gain control in the event of an LE/VSE software condition.
- If an LE/VSE- or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs, the CICS thread is terminated. This occurs even if a CICS HANDLE ABEND is active, because CICS HANDLE ABEND does not gain control in the event of an LE/VSE abend.
- If a user abend or program check occurs, the following actions take place:

  If no EXEC CICS HANDLE ABEND is active, and TRAP(ON) is set in the child enclave, LE/VSE condition handling takes place. If the abend or program check remains unhandled, the problem is not propagated to the parent enclave. The CICS thread is terminated with an abend.

  An active EXEC CICS HANDLE ABEND overrides the setting of TRAP. The action defined by the EXEC CICS HANDLE ABEND takes place.

Creating Child Enclaves Using the C system() Function

Child enclaves created by the C `system()` function get run-time options through a merge from the usual sources (see Chapter 5, “Using Run-Time Options,” on page 33 for more information). Therefore, you can set run-time options on an enclave-by-enclave basis. See *LE/VSE C Run-Time Library Reference* and *LE/VSE C Run-Time Programming Guide* for information on the `system()` function.

When you perform a `system()` function to a COBOL program, in the form:

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

the run-time options specified in the PARM= portion of the `system()` function are ignored. However, run-time options are merged from CEEDOPT, CEEUOPT, and the CEEAUE_OPTION from the assembler user exit.

How Conditions Arising in Child Enclaves Are Handled

Condition handling varies depending on the source of the condition, and the settings of the TRAP run-time option in the parent and child enclaves.

If a software condition occurs, LE/VSE condition handling takes place. If the condition remains unhandled, the behavior depends upon the severity of the condition, and the settings of the TRAP run-time option in the parent and child enclaves. The following conditions might cause the child enclave to terminate:

- Unhandled user abend
- Unhandled program check

Table 59 summarizes the different types of behavior that can occur when an unhandled condition occurs in a child enclave created by the C `system()` function.

Table 59. Unhandled Condition Behavior in a system()-Created Child Enclave

	Parent Enclave TRAP(ON) Child Enclave TRAP(ON)	**Parent Enclave TRAP(ON) Child Enclave TRAP(OFF)**	**Parent Enclave TRAP(OFF) Child Enclave TRAP(ON)**	**Parent Enclave TRAP(OFF) Child Enclave TRAP(OFF)**
Unhandled condition severity 0 or 1	Resume child enclave	Resume child enclave	Resume child enclave	Resume child enclave
Unhandled condition severity 2 or above	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition	Resume parent enclave, and ignore condition
Non-LE/VSE abend	Resume parent enclave, and ignore condition	Process terminated with VSE system abend message or LE/VSE message CEE3322C	Resume parent enclave, and ignore condition	Process terminated with VSE system abend message or LE/VSE message CEE3322C
Program check	Resume parent enclave, and ignore condition	Process terminated with VSE system abend message	Resume parent enclave, and ignore condition	Process terminated with VSE system abend message

If an LE/VSE- or CEEBXITA-initiated (generated by setting the CEEAUE_ABND field of CEEBXITA) abend occurs in a child enclave created by a call to `system()`, the entire process is terminated.

Other Nested Enclave Considerations

The following sections contain other information you might need to know when creating nested enclaves. The topics include:

- The string that CEE5PRM returns for each type of child enclave (see *LE/VSE Programming Reference* for more information about the CEE5PRM callable service)
- The return and reason codes that are returned on termination of the child enclave
- How the assembler user exit handles nested enclaves
- MSGFILE considerations
- AMODE considerations

What the Enclave Returns from CEE5PRM

CEE5PRM returns to the calling routine the user parameter string that was specified at program invocation. Only program arguments are returned.

See Table 60 to determine whether a user parameter string was passed to your routine, and where the user parameter string is found. This depends on the method you used to create the child enclave, the language of the routine in the child enclave, and the PLIST or SYSTEM setting of the main routine in the child enclave. If a user parameter string was passed to your routine, the user parameter string is extracted from the command-line equivalent for your routine (shown in Table 61 on page 397) and returned to you.

Note: Under CICS, CEE5PRM always returns a blank string.

Table 60. Determining the Command-Line Equivalent

Language	Option	Suboption	Parameter String
C	`#pragma runopts(PLIST)`	HOST	OS
		Parameter string from the command string passed to `system()`	Not available
PL/I	SYSTEM compiler option	VSE	CICS, DLI
		Parameter string from the command string passed to `system()`	Not available
COBOL	Null		
LE/VSE-conforming assembler	CEEENTRY PLIST parameter	HOST	OS
		Parameter string from the command string passed to `system()`	Not available

If Table 60 indicates that a parameter string was passed to your routine at invocation, the string is extracted from the command-line equivalent listed in the right-hand column of Table 61 on page 397. The command-line equivalent depends on the language of your routine and the run-time options specified for it.

Table 61. Determining the Order of Run-Time Options and Program Arguments

Language of Routine	Run-Time Options in Effect?	Order of Run-Time Options and Program Arguments
C	`#pragma runopts(EXECOPS)` is specified (or defaulted)	`#pragma runopts(NOEXECOPS)` is specified
	run-time options / user parms	entire string is user parms
COBOL	CBLOPTS(ON)	CBLOPTS(OFF)
	run-time options are not available; entire string is user parms	run-time options are not available; entire string is user parms
PL/I	PROC OPTIONS(NOEXECOPS) is not specified	PROC OPTIONS(NOEXECOPS) is specified
	run-time options / user parms	entire string is user parms
LE/VSE-conforming assembler	CEEENTRY EXECOPS=ON is specified	CEEENTRY EXECOPS=OFF is specified
	run-time options / user parms	entire string is user parms

Finding the Return and Reason Code from the Enclave

The following list tells where to look for the return and reason codes that are returned to the parent enclave when a child enclaves terminates:

- EXEC CICS LINK or EXEC CICS XCTL

  If the CICS thread was not terminated, the return code is placed in the optional RESP2 field of EXEC CICS LINK or EXEC CICS XCTL. The reason code is discarded.
- C's `system()` function

  If the target command or program of `system()` cannot be started, "-1"is returned as the function value of `system()`. Otherwise, the return code of the child enclave is reported as the function value of `system()`, and the reason code is discarded. (See *LE/VSE C Run-Time Library Reference* and *LE/VSE C Run-Time Programming Guide* for more information about the `system()` function.)

Assembler User Exit

An assembler user exit (CEEBXITA) is driven for enclave initialization and enclave termination regardless of whether the enclave is the first enclave created in the process or a nested enclave. The assembler user exit differentiates between first and nested enclave initialization.

MSGFILE Considerations

You can specify the same message file across nested enclaves; LE/VSE coordinates the use of the same filename across nested enclaves. If you specify a different MSGFILE filename in each enclave, LE/VSE honors each filename.

The message file is not closed when control returns from a child enclave. All open message files are closed during process termination.

AMODE Considerations

ALL31 should have the same setting for all enclaves within a process. You cannot invoke a nested enclave that requires ALL31(OFF) from an enclave running with ALL31(ON).

Part 6. Appendixes

© Copyright IBM Corp. 1991, 2005

Appendix A. Guidelines for Writing Callable Services

If you want to write services similar in form and description to LE/VSE callable services, follow the guidelines listed in this chapter.

- Callable service parameters must follow the data type descriptions outlined in *LE/VSE Programming Reference*.
- Argument passing is by one level of indirection, either "by reference" or "by value". See "Passing Arguments between Routines" on page 50 for these argument passing styles.
- Avoid the use of operating system services and macros. Use LE/VSE services whenever possible.
- Always use the prototype definition or the entry declaration whenever possible.
- Avoid using the CEE5SPM callable service (see *LE/VSE Programming Reference*). CEE5SPM can change the condition handling semantics of the HLLs supported by LE/VSE.
- LE/VSE assumes the following defaults for character strings:
  - For input arguments, a length-prefixed string (with the length of the 2-byte prefix not included in the length value)
  - For output arguments, a fixed-length string of 80 bytes, padded on the right with blanks as necessary.
- Allow a feedback code area to be optionally passed as the last parameter to the callable service. The feedback code must be a FEED_BACK data type and conform to the layout described in Chapter 14, "Using Condition Tokens," on page 171.
- If omitted arguments are permitted by the HLL, a zero or NULL pointer must be used to indicate the omitted parameter in the parameter list that is passed to the callable service. For example:

address of parm1
address of parm2
0

  The last parameter passed in the list must have the high-order bit on to indicate that it is last. If the last parameter is omitted, the zero value that the user passes in the parameter list must have the high-order bit on, for example X'80000000'. Therefore, you must allow the user of the callable service to check for this bit when the last parameter passed to the service is omitted.
- When documenting callable services, follow the same general format used to document each of the callable services in this book. Each callable service description should contain (in this order):
  - A general description of what the service does
  - A diagram indicating the syntax of the call to the service
  - A complete description of each callable service parameter and an identification of the required data type
  - A list of possible feedback codes that can be returned by the service to its caller

© Copyright IBM Corp. 1991, 2005

- Usage notes that provide additional needed information to the user, such as a list of related callable services
- An example or examples of usage.

Appendix B. Using Operating System and Subsystem Parameter List Formats

This appendix describes the various formats of parameters passed to and from operating systems and subsystems. In most cases you do not need to know these formats in order to pass or receive parameters in your application. For cases in which you want to directly access the parameter list that is passed, the format and contents of the parameter list are shown below.

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

- "C Parameter Passing Considerations"
- "COBOL Parameter Passing Considerations" on page 406
- "PL/I Main Procedure Parameter Passing Considerations" on page 406

C Parameter Passing Considerations

C generally supports a single character string as a parameter to a main routine. It parses the string into tokens that are accessed by the `argc` and `argv` parameters of the main function.

In addition, C supports various other styles of passing a set of parameters to the main routine. For example, you can pass a parameter to a main routine as a single value, a pointer to a value, or a pointer to a list of values. In these cases, the set of parameters is not parsed. It is assumed that the invoker of the application (for example, the operating system) has stored the address of the set of parameters in register 1 prior to entry into the main routine. Depending on how the parameters are passed, register 1 points on entry to the entities illustrated in Figure 128:

```
Style 1: Register 1 contains parameter value (restricted to integer types)

     Register 1 = parameter value


Style 2: Register 1 contains pointer to parameter value

     Register 1 = pointer → parameter value


Style 3: Register 1 contains pointer to array
         of pointers to parameter values

     Register 1 = pointer → (pointer0→value0)
                              (pointer1→value1)
                              (pointer2→value2)
                                       .
                                       .
                                       .
                              (pointern→valuen)
```

Figure 128. Some C Parameter Passing Styles

© Copyright IBM Corp. 1991, 2005

When you code your C main routine, you elect to use one of the styles shown in Figure 128 on page 403 by specifying the PLIST(OS) run-time option in `#pragma runopts` (see “C PLIST and EXECOPS Interactions” on page 405). When PLIST(OS) is specified, C makes the parameter list available through a pair of macros: `__R1` and `__osplist`. Use the macros in your main routine according to which parameter list style your routine receives:

__R1 of type `void *`
: __R1 contains the value that is in register 1 on entry into the main routine. It provides access to the parameters when they are passed according to the first two styles shown in Figure 128 on page 403.

__osplist of type `void **`
: `__osplist` acts as an array of pointers to parameters. It is derived from `__R1` and provides access to the parameters when they are passed according to the third style shown in Figure 128 on page 403.

You must include the header file `stdlib.h` when using `__R1` or `__osplist`.

Figure 129 illustrates how these macros can be used to access items in the three alternative parameter arrangements.

```
Style 1:

     Register 1 = __R1


Style 2:

     Register 1 = __R1 → *__R1


Style 3:

     Register 1 = __R1 →(__osplist[0]→*__osplist[0])
                         (__osplist[1]→*__osplist[1])
                         (__osplist[2]→*__osplist[2])
                                            .
                                            .
                                            .
                         (__osplist[n]→*__osplist[n])
```

Figure 129. Accessing Parameters Using Macros __R1 and __osplist

Suitable casting and dereferencing are required when using these macros, as shown in Figure 130 on page 405, according to the parameter passing style in use.

```
Style 1:

     parm     = (int) __R1;   (restricted to integer types)


Style 2:

     parm_ptr = (float *) __R1;
     parm     = &asterisk ((float *) __R1);


Style 3:

     parm0_ptr = (float *) __osplist[0];
     parm0     = &asterisk ((float *) __osplist[0]);
```

Figure 130. Examples of Casting and Dereferencing

The third parameter passing style is also supported by certain invoker-specific macros and functions (for example, `__pcblist` and `__csplist` for invokers of DL/I and Cross System Product). `__osplist` is a generalized form of the more specialized `__pcblist` and `__csplist` macros; it can be used in their place or in cases where they do not apply.

C PLIST and EXECOPS Interactions

You can use C `#pragma runopts` to specify to the C compiler a list of options to be used at run time. Two of the options of `#pragma runopts` affect the format of the argument list passed to the application on initialization: EXECOPS and PLIST.

EXECOPS allows you to specify run-time options PARM parameter of the JCL EXEC statement at application invocation. NOEXECOPS indicates that run-time options cannot be so specified. When the EXECOPS run-time option is specified, LE/VSE removes any run-time options that are present before passing the parameter list to the main routine.

PLIST indicates in what form the invoked routine should expect the argument list. You can specify PLIST with the following values under LE/VSE:

HOST The argument list is assumed to be a character string. LE/VSE uses the standard VSE parameter list format; a string prefixed by a halfword length field.

OS The inbound parameter list is assumed to be in the OS linkage format in which register 1 points to a parameter address list. No run-time options are available. Register 1 is not interrogated by LE/VSE.

The EXECOPS, NOEXECOPS, and PLIST options can alter the format of the argument list passed to your application, depending on the combination of options specified. The setting of EXECOPS determines whether LE/VSE looks for run-time parameters in the inbound parameter list. The effects of the interactions of these options under the various operating environments are summarized in Table 62 on page 406

Table 62. Interactions of C PLIST and EXECOPS (`#pragma runopts`)

Operating Environment	Method of Invocation	PLIST Suboption	EXECOPS	argc/argv	__R1/__osplist and PCBs
Batch	EXEC PGM=, PARM= <run-time options> / <user args>	HOST	Yes (default). <run-time options> honored	argc = number of tokenized args in <user args> argv[0...argc-1] = tokenized args in <user args>	
Batch	EXEC PGM=, PARM= <run-time options> / <user args>	HOST	No. <run-time options> ignored	argc = number of tokenized args in the entire PARM string, that is, <run-time options> / <user args> argv[0...argc-1] = tokenized args in the entire PARM string	
Batch	Driver link to C main passing noncharacter parameter list	OS	n/a	argc=1 argv[0] = name of C main routine	Access register 1 through __osplist macro as defined in stdlib.h
Batch DL/I	DL/I invokes C main routine	OS Specify ENV(DLI) also.	n/a	argc=1 argv[0] = 0	Access PCBs through C macros as defined in ims.h
CICS	CICS invokes C main routine	n/a	n/a	argc=1 argv[0] = transaction id	

COBOL Parameter Passing Considerations

When COBOL is the main routine, LE/VSE sets the argument list passed to the application on initialization as follows:

VSE

- If the COBOL main is invoked in a non-CICS environment:

  A halfword prefixed string is passed to the application after run-time options have been removed.
- If the COBOL main is invoked from an assembler routine using standard assembler linkage conventions, then Register 1 and the argument list are passed without change.

CICS

- If the COBOL main is invoked in a CICS environment, Register 1 is passed without change.

PL/I Main Procedure Parameter Passing Considerations

The format of the parameter list passed to a PL/I main procedure from the operating system is controlled by the SYSTEM compiler option and also by options on the main PROCEDURE statement.

The SYSTEM compiler option specifies the format used to pass parameters to the PL/I main procedure, and indicates the operating environment under which the program runs: VSE batch without DL/I, VSE batch with DL/I, or CICS.

The NOEXECOPS procedure option indicates that run-time options are not present in the operating system parameter list. The NOEXECOPS option can be explicitly specified or implicitly defaulted. Otherwise, it is assumed that run-time options might be present in the operating system parameter list. If present, these run-time options are removed by run-time initialization before the PL/I main procedure gains control.

In order for run-time options to be passed in the operating system parameter list for SYSTEM(VSE), the PL/I main procedure must receive no parameters or receive a single parameter that is a varying character string. If this is not the case, NOEXECOPS is always defaulted.

The OPTIONS(BYVALUE) or OPTIONS(BYADDR) procedure options indicate whether the main procedure parameters are passed directly or indirectly. If SYSTEM(CICS) is specified for a PL/I main procedure, the OPTIONS(BYVALUE) procedure option is defaulted at compilation time; OPTIONS(BYADDR) is not permitted. If SYSTEM(DLI) is specified for a PL/I main procedure, the OPTIONS(BYADDR) procedure option is defaulted at compilation time; OPTIONS(BYVALUE) is not permitted. See “Passing Arguments between Routines” on page 50 for additional information about LE/VSE parameter passing.

The following table describes the interaction of the PL/I SYSTEM and NOEXECOPS options. Their effect is described in terms of the parameters that are coded on the main procedure statement and also the incoming system, subsystem or assembler parameter list as initially received by LE/VSE.

Table 63. Interactions of SYSTEM and NOEXECOPS

SYSTEM Setting	No Run-Time Options (NOEXECOPS)	Run-Time Options May be Present
SYSTEM(VSE)	If the main procedure parameter is a single varying character string, a VSE parameter list is assumed and repackaged so the main procedure receives a halfword prefixed string. The entire string is passed to the main procedure without change. Otherwise, the parameter list is passed without change.	If the main procedure parameter is a single varying character string, a VSE parameter list is assumed and repackaged so the main procedure receives a halfword prefixed string. Any run-time options are removed from the string, and the (potentially) altered string is passed. Otherwise, the parameter list is passed without change.
SYSTEM(DLI)	The parameter list is passed without change.	Not allowed
SYSTEM(CICS)	The parameter list is passed without change.	Not allowed

Note:
NOEXECOPS is always implied for SYSTEM(DLI) and SYSTEM(CICS). NOEXECOPS is also implied for SYSTEM(VSE) if the main procedure has more than one parameter or a single parameter that is not a varying character string.

Appendix C. Sort and Merge Considerations

This chapter discusses the run-time aspects of sort and merge operations. For details on the compile-time aspects of sort and merge, including instructions on coding the sort and merge procedures, see your compiler programming guide.

Understanding the Basics

Under LE/VSE, you can invoke the sort facility to sort or merge records in a particular sequence. A sort operation takes an unordered sequence of input data, arranges it according to a specified key or pattern, and places it into an output file. A merge operation compares two or more files that have already been sorted according to an identical key and combines them in a specified order in an output file.

To invoke the sort facility in LE/VSE, you can use either of the following:

- An HLL construct
  - COBOL's SORT and MERGE verbs
  - PL/I's PLISRT*x* interface, where *x* is replaced by A, B, C, or D

    You cannot call the PLISRT*x* interface under CICS.
- A method other than an HLL construct (for example, assembler routines or JCL).

In the batch environment, your IBM sort/merge licensed program must be DFSORT/VSE or an equivalent that honors the DFSORT/VSE parameter list. Whenever DFSORT/VSE is mentioned in this chapter, you can use any equivalent SORT product.

Invoking DFSORT/VSE Directly

For information about using the methods to run DFSORT/VSE directly with JCL or to invoke DFSORT/VSE directly from an assembler program, see *DFSORT/VSE Application Programming Guide* Also see that book for details on the many DFSORT/VSE built-in features that you can use to eliminate the need for writing program logic (for example, the INCLUDE, OMIT, OUTREC, and SUM statements).

Using the COBOL SORT and MERGE Verbs

This section contains a high-level overview of COBOL SORT and MERGE verbs. It is designed to introduce you to concepts that help you understand some of the special considerations for using these COBOL statements in LE/VSE. For a detailed description of how to use SORT and MERGE, see *IBM COBOL for VSE/ESA Programming Guide*

A COBOL routine that contains a sort operation can be organized so that an *input procedure* reads and operates on one or more input files before the files are actually sorted:

```
SORT...INPUT PROCEDURE
```

You can also specify an *output procedure* that processes the files after they are sorted:

```
SORT...OUTPUT PROCEDURE
```

© Copyright IBM Corp. 1991, 2005

These input and output procedures can be used to add, delete, alter, edit, or otherwise modify the records.

You can also sort records under COBOL without any processing by the input and output procedures. For example, to read records into a new file for sorting without any preliminary processing, specify:

```
SORT...USING
```

To transfer sorted records to a file without any further processing, specify:

```
SORT...GIVING
```

User Exit Considerations

SORT or MERGE COBOL verbs can trigger COBOL-generated user exits (E15 for sort, E35 for merge). These exits include any input or output procedures. However, the exits are not triggered when a COBOL USING or GIVING statement is in effect and the files qualify for FASTSRT.

ILC is permitted within the DFSORT/VSE user exits if ILC is permitted among programs within the same phase. ILC is not permitted in a DFSORT/VSE user exit if it involves dynamically loaded routines.

Condition Handling Considerations

This section summarizes how LE/VSE condition handling behaves when a COBOL/VSE or VS COBOL II routine is involved in a SORT or MERGE.

Program Interrupts

User handlers established by the routine that initiated the SORT/MERGE are able to handle program interrupts as they are presented to the condition manager by a condition token. Normal condition handling as described in Chapter 11, “LE/VSE Condition Handling Introduction,” on page 103 occurs.

Establishment of HLL-specific handlers and user handlers is not supported while in a SORT input or output procedure. The results are unpredictable, and the condition handler does not attempt to diagnose this case.

HLL-specific handlers and user handlers established by the routine **called by** an input or output procedure are able to handle program interrupts. However, because these exits are typically invoked many times (equivalent to the number of records being sorted for each exit), it is recommended that you register the handler within the application that initiated the SORT/MERGE in order to save overhead.

LE/VSE-Signaled Conditions

HLL-specific handlers and user handlers established by the routine that initiated the SORT/MERGE are able to handle any condition signaled by LE/VSE. Normal condition handling as described in Chapter 11, “LE/VSE Condition Handling Introduction,” on page 103 occurs.

Abends

When there is an abend, LE/VSE condition handling behavior depends on where the abend occurred, and the run-time options that are in effect:

- If the TRAP(ON) run-time option is in effect, the LE/VSE condition handler intercepts the abend. The condition handler then gives control to the DFSORT/VSE STXIT exit if:
  - The condition handler determines that the abend occurred with DFSORT/VSE

– The DFSORT/VSE STXIT exit has been established
– No overriding STXIT exit has been established (such as the STXIT exit established by DL/I, or the STXIT exit established by DB2 running in single-user mode)

The DFSORT/VSE STXIT exit then performs various cleanups and recoveries, produces informational dumps and messages, as appropriate. The task is then terminated without LE/VSE condition handling being invoked.

If the LE/VSE condition handler determines that the abend did not occur within DFSORT/VSE, or that the DFSORT/VSE STXIT exit has not been established, or that an overriding STXIT exit has been established, normal condition handling as described in Chapter 11, “LE/VSE Condition Handling Introduction,” on page 103 occurs. For information about condition handling behavior when your application uses DL/I, see Chapter 24, “Running Applications with DL/I,” on page 313. For information about condition handling behavior when your application uses DB2, see Chapter 23, “Running Applications with DB2,” on page 311.

- If the TRAP(ON) run-time option is not in effect, the DFSORT/VSE STXIT exit, if it has been established, intercepts the abend. The DFSORT/VSE STXIT exit then performs various cleanups and recoveries, produces informational dumps and messages, as appropriate. The task in then terminated without LE/VSE condition handling being invoked.

Using the PL/I PLISRTx Interface

This section contains a high-level overview of the PLISRT*x* interfaces to DFSORT/VSE. It is designed to introduce you to concepts that help you understand some of the special considerations for using these PL/I interfaces in LE/VSE. For a detailed description of how to use PLISRT*x*, see *IBM PL/I for VSE/ESA Programming Guide*

PL/I provides a SORT interface called PLISRT*x*. When you make a call to PLISRT*x*, you replace *x* with A, B, C, or D, depending on whether your input comes from a file or a PL/I subroutine, and whether your output is to be written to a data set or processed by a PL/I subroutine:

PLISRTA Unsorted input is read from a file and then sorted. The sorted output is written to a file.

PLISRTB Unsorted input is provided and processed by a PL/I subroutine before sorting. The sorted output is written to a file.

PLISRTC Unsorted input is read from a file and then sorted. The sorted output is then processed in some way by a PL/I subroutine.

PLISRTD Unsorted input is provided and processed by a PL/I subroutine before sorting. The sorted output is then processed by a PL/I subroutine.

In the call to PLISRT*x*, you also pass information about your data, using the SORT and RECORD arguments, and specify the maximum amount of storage you will allow DFSORT/VSE to use.

User Exit Considerations

Your input handling subroutine and output handling subroutine must be written in PL/I. PL/I generates a DFSORT/VSE E15 exit for your input handling subroutine and a DFSORT/VSE E35 exit for your output handling subroutine.

A call to one of the PLISRT*x* interfaces might trigger a call to user exit E15 or E35, depending on whether a subroutine processed your input before sorting, and output after sorting, as shown in Table 64.

Table 64. DFSORT/VSE Exit Called as a Function of a PLISRTx Interface Call

PL/I Sort Interface	DFSORT/VSE Exit
PLISRTA	None
PLISRTB	E15
PLISRTC	E35
PLISRTD	E15 and E35

ILC is not supported within SORT exits when the PLISRT*x* interface is used from a PL/I routine.

Condition Handling Considerations

Input and output handling subroutines can issue GOTOs. If you need to deactivate the SORT program for any reason while in one of these exits, issue a GOTO out of the subroutine.

Program Interrupts and LE/VSE-Signaled Conditions

PL/I ON-units can be established in any of the following:
- The routine that made a call to PLISRT*x*
- The input(E15) or output(E35) procedure
- A routine called by the input or output procedure

These ON-units can handle program interrupts and LE/VSE-signaled conditions. Normal condition handling, as described in Chapter 11, “LE/VSE Condition Handling Introduction,” on page 103, occurs.

Abends

When there is an abend, LE/VSE condition handling behavior depends on where the abend occurred, and the run-time options that are in effect:
- If the TRAP(ON) run-time option is in effect, the LE/VSE condition handler intercepts the abend. The condition handler then gives control to the DFSORT/VSE STXIT exit if:
  - The condition handler determines that the abend occurred with DFSORT/VSE
  - The DFSORT/VSE STXIT exit has been established
  - No overriding STXIT exit has been established (such as the STXIT exit established by DL/I, or the STXIT exit established by DB2 running in single-user mode)

  The DFSORT/VSE STXIT exit then performs various cleanups and recoveries, produces informational dumps and messages, as appropriate. The task is then terminated without LE/VSE condition handling being invoked.

  If the LE/VSE condition handler determines that the abend did not occur within DFSORT/VSE, or that the DFSORT/VSE STXIT exit has not been established, or that an overriding STXIT exit has been established, normal condition handling as described in Chapter 11, “LE/VSE Condition Handling Introduction,” on page 103 occurs. For information about condition handling behavior when your application uses DL/I, see Chapter 24, “Running Applications with DL/I,” on page 313. For information about condition handling behavior when your application uses DB2, see Chapter 23, “Running Applications with DB2,” on page 311.

- If the TRAP(ON) run-time option is not in effect, the DFSORT/VSE exit, if it has been established, intercepts the abend. The DFSORT/VSE STXIT exit performs various cleanups and recoveries, produces informational dumps and messages, as appropriate. The task in then terminated without LE/VSE condition handling being invoked.

  Note: If, in the call to PLISRT*x*, you specify the CKPT parameter in the SORT argument, LE/VSE temporarily disables normal LE/VSE condition handling during DFSORT/VSE phase 3 processing. Any abend that occurs during phase 3 is handled by DFSORT/VSE.

Storage Use during a Sort or Merge Operation

In general, the more storage DFSORT/VSE has available, the faster the sorting operation is performed. Certain parameters specified during the installation of DFSORT/VSE determine the amount of storage used during its operation.

DFSORT/VSE does not use GETVIS storage. Sufficient program storage must be reserved for:

- Loading the phase specified on the EXEC statement
- DFSORT/VSE modules
- DFSORT/VSE input and output buffers
- DFSORT/VSE working storage

Note: Program storage may be reserved by using the JCL EXEC statement SIZE parameter. Your run-time JCL should look like:

```
// EXEC pgrmid,SIZE=(pgrmid,nnnK)
```

where *nnnK* is the amount of program storage required for DFSORT/VSE.

GETVIS storage must be reserved for:

- Application programs that are dynamically loaded
- LE/VSE library routines
- LE/VSE stack and heap storage

COBOL users can override the DFSORT/VSE parameter values specified at installation. The STORAGE keyword on the DFSORT/VSE control statement, or the COBOL SORT-CORE-SIZE special register, can be used for this purpose. (For the meaning of this key word see *DFSORT/VSE Application Programming Guide* .)

Note: Be careful not to override the storage allocation to the extent that more than the reserved program storage is used for sort's operation.

Sorting under CICS

Under CICS, you can invoke the sort facility from a COBOL/VSE program. You can use the COBOL SORT statement (along with a sort program that runs under CICS) to sort small amounts of data. The format of the parameter list passed to the sort program under CICS is shown in figure Figure 131 on page 414.

Offset	Field
0(0)	CONTROL
4(4)	IP_PROC
8(8)	OP_PROC
12(C)	RESERVED
16(10)	ALTSEQ
20(14)	RESERVED
32(20)	END_MARK

Figure 131. Format of Sort Parameter List under CICS

CONTROL
The address of the sort control statements, or zero.

IP_PROC
The address of the input procedure specified in the INPUT PROCEDURE phrase of the SORT statement.

OP_PROC
The address of the output procedure specified in the OUTPUT PROCEDURE phrase of the SORT statement.

ALTSEQ
The address of the collating-sequence table specified in the COLLATING SEQUENCE phrase of the SORT statement, or zero.

END_MARK
A fullword containing X'FFFFFFFF', indicating the end of the parameter list.

For more information about using the COBOL SORT statement under CICS, see *IBM COBOL for VSE/ESA Programming Guide*

Appendix D. LE/VSE Macros

The macros identified in this appendix are provided by LE/VSE as programming interfaces for customers.

- CEECAA (see "CEECAA Macro— Generate a CAA Mapping" on page 344)
- CEECIB, a macro for generating a CIB mapping (refer to the section "Debugging with the Condition Information Block" in the *LE/VSE Debugging Guide and Run-Time Messages*).
- CEEDSA (see "CEEDSA Macro— Generate a DSA Mapping" on page 345)
- CEEENTRY (see "CEEENTRY Macro— Generate an LE/VSE-Conforming Prolog" on page 341)
- CEEFETCH (see "CEEFETCH Macro— Dynamically Load a Routine that Can Be Later Deleted" on page 350)
- CEELOAD (see "CEELOAD Macro— Dynamically Load a Routine" on page 348)
- CEEPPA (see "CEEPPA Macro— Generate a PPA" on page 345)
- CEERELES (see "CEERELES Macro— Dynamically Delete a Routine" on page 353)
- CEETERM (see "CEETERM Macro— Terminate an LE/VSE-Conforming Routine" on page 343)
- CEEXPIT (see "Macros that Generate the PIPI Table" on page 365)
- CEEXPITY (see "Macros that Generate the PIPI Table" on page 365)
- CEEXPITS (see "Macros that Generate the PIPI Table" on page 365)
- `__csplist` (see "C Parameter Passing Considerations" on page 403)
- `__osplist` (see "C Parameter Passing Considerations" on page 403)
- `__pcblist` (see "C Parameter Passing Considerations" on page 403)
- `__R1` (see "C Parameter Passing Considerations" on page 403)

© Copyright IBM Corp. 1991, 2005

Language Environment Glossary

A

abend. Abnormal end of application.

absolute value. The magnitude of a real number regardless of its algebraic sign.

active routine. The currently executing routine.

additional heap. An LE/VSE heap created and controlled by a call to CEECRHP. See also *below heap*, *anywhere heap*, and *initial heap*.

addressing mode. An attribute that refers to the address length that a routine is prepared to handle upon entry. Addresses may be 24 or 31 bits long.

aggregate. A structured collection of data items that form a single data type. Contrast with *scalar*.

American National Standard Code for Information Interchange (ASCII). The code developed by the American National Standards Institute (ANSI) for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

AMODE. Addressing mode.

anywhere heap. The LE/VSE heap controlled by the ANYHEAP run-time option. It contains library data, such as LE/VSE control blocks and data structures not normally accessible from user code. The anywhere heap may reside above 16MB. See also *below heap*, *additional heap*, and *initial heap*.

application. A collection of one or more routines cooperating to achieve particular objectives.

application program. A collection of software components used to perform specific types of work on a computer, such as a program that does inventory control or payroll.

argument. An expression used at the point of a call to specify a data item or aggregate to be passed to the called routine.

array. An aggregate that consists of data objects, each of which may be uniquely referenced by subscripting.

array element. A data item in an array.

ASCII. American National Standard Code for Information Interchange.

Asian date format. In this book, Asian date format refers to the era picture strings associated with the Japanese or Chinese eras. Era picture strings begin with a less than character < and end with a greater than character >. The characters inside are either capital Js or Cs.

assembler. see *High Level Assembler*.

automatic call library. Contains object modules that are to be used as secondary input to the linkage editor to resolve external symbols left undefined after all the primary input has been processed.

The automatic call library may be:

- Sublibraries containing object modules, with or without linkage editor control statements
- The sublibrary containing LE/VSE run-time routines (PRD2.SCEEBASE or PRD2.SCEECICS)

automatic data. Data that does not persist across calls to other routines. Automatic data may be automatically initialized to a certain value upon entry and reentry to a routine.

automatic storage. Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

B

batch. Pertaining to activity involving little or no user action. Contrast with *interactive*.

below heap. The LE/VSE heap controlled by the BELOWHEAP run-time option, which contains library data, such as LE/VSE control block and data structures not normally accessible from user code. Below heap always resides below 16MB. See also *anywhere heap*, *initial heap*, and *additional heap*.

buffer. An area of storage into which data is read or from which it is written. Typically, buffers are used only for temporary storage.

by content. See *pass by content*.

by reference. See *pass by reference*.

by value. See *pass by value*.

C

callable services. A set of services that can be invoked by an LE/VSE-conforming high level language using

© Copyright IBM Corp. 1991, 2005

the conventional LE/VSE-defined call interface, and usable by all programs sharing the LE/VSE conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

callable service stub. Contains addressing code to access LE/VSE callable service routines.

callee. Receiver of a call.

caller. A routine that calls another routine.

century window. The 100-year interval in which LE/VSE assumes all 2-digit years lie. The LE/VSE default century window begins 80 years before the system date.

chained list. Synonym for *linked list*.

child enclave. The *nested enclave* created as a result of certain commands being issued from a *parent enclave*.

CICS. Customer Information Control System.

CICS run unit. Consists of a statically and/or dynamically bound set of one or more phases which can be loaded by a CICS loader. A CICS run unit is equivalent to an LE/VSE *enclave*.

CICS translator. A routine that accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source.

COBOL. COmmon Business-Oriented Language. A high level language, based on English, that is primarily used for business applications.

COBOL run unit. A COBOL-specific term that defines the scope of language semantics. Equivalent to an LE/VSE *enclave*.

command line. The command used to invoke an application program, and the associated program arguments and LE/VSE run-time options. This can be the job control EXEC statement and the associated PARM parameter, or the parameter string passed to the C `system()` function.

COMMAREA. A communication area made available to applications running under CICS.

compilation unit. An independently compilable sequence of HLL statements. Each HLL product has different rules for what makes up a compilation unit. Synonym for *program unit*.

condition. An exception that has been enabled, or recognized, by LE/VSE and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

condition handler. A user-written condition handler or language-specific condition handler (such as a PL/I ON-unit) invoked by the LE/VSE *condition manager* to respond to conditions.

condition handling. In LE/VSE, the diagnosis, reporting, and/or tolerating of errors that occur in the run-time environment.

condition manager. Manages conditions in the common execution environment by invoking various user-written and language-specific *condition handlers*.

condition step. The step of the LE/VSE condition handling model that follows the enablement step. In the condition step, user-written condition handlers and PL/I ON-units are first given a chance to handle a condition. See also *enablement step* and *termination imminent step*.

condition token. In LE/VSE, a data type consisting of 96 bits (12 bytes). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

constructed reentrancy. The attribute of applications that contain external data and require additional processing to make them reentrant. Contrast with *natural reentrancy*.

Customer Information Control System (CICS). CICS is an OnLine Transaction Processing (OLTP) system that provides specialized interfaces to databases, files and terminals in support of business and commercial applications.

D

data type. The properties and internal representation that characterize data.

DBCS. Double-byte character set.

decimal overflow. A condition that occurs when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the result.

default. A value that is used when no alternative is specified.

direct parameter passing. Placing a value directly in the parameter list body.

disabled/enabled. See *enabled/disabled*.

double-byte character set (DBCS). A collection of characters represented by a two-byte code.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision. See also *precision* and *single-precision*.

doubleword. A sequence of bits or characters that comprises two computer words and can be addressed as a unit.

DSA. Dynamic storage area.

dynamic call. A call that results in the resolution of the called routine at run time. Contrast with *static call*.

dynamic storage. Storage acquired as needed at run time. Contrast with *static storage*.

dynamic storage area (DSA). An area of storage obtained during the running of an application that consists of a register save area and an area for automatic data, such as program variables. DSAs are generally allocated within LE/VSE-managed stack segments. DSAs are added to the stack when a routine is entered and removed upon exit in a last in, first out (LIFO) manner. In LE/VSE, a DSA is known as a *stack frame*.

E

EBCDIC. Extended binary-coded decimal interchange code.

EIB. EXEC interface block.

enabled/disabled. A condition is enabled when its occurrence will result in the execution of condition handlers or in the performance of a standard system action to handle the condition as defined by LE/VSE.

A condition is disabled when its occurrence will apparently be ignored by the condition manager.

enablement. The determination by a language at run time that an exception should be processed as a condition. This is the capability to intercept an exception and to determine whether it should be ignored or not; unrecognized exceptions are always defined to be enabled. Normally, enablement is used to supplement the hardware for capabilities that it does not have and for language enforcement of the language's semantics. An example of supplementing the hardware is the specialized handling of floating-point overflow exceptions based on language specifications (on some machines this can be achieved through masking the exception).

enablement step. The first step of the LE/VSE condition handling model. In the enablement step it is determined whether an exception is to be *enabled* and processed as a condition. See also *condition step* and *termination imminent step*.

enclave. In LE/VSE, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

entry name. In assembler language, a programmer-specified name within a control section that identifies an entry point and can be referred to by any control section.

entry point. In assembler language, the address or label of the first instruction that is executed when a routine is entered for execution.

environment. A set of services and data available to a program during execution. In LE/VSE, environment is normally a reference to the run-time environment of HLLs at the enclave level.

epilog. Code generated at the end of a routine, normally causing a return to the caller of the routine.

ESDS. Entry sequenced data sets. See *VSAM*.

EXEC interface block (EIB). In CICS, a control block containing information useful in the execution of an application, such as a transaction identifier and a time and a date when the transaction is started.

execution time. Synonym for *run time*.

execution environment. Synonym for *run-time environment*.

extended binary-coded decimal interchange code (EBCDIC). A set of 256 eight-bit characters.

external data. Data that persists over the lifetime of an enclave and maintains last-used values whenever a routine within the enclave is reentered. Within an enclave consisting of a single phase, it is equivalent to COBOL external data.

external routine. A procedure or function that may be invoked from outside the program in which the routine is defined.

F

facility ID. A string of three characters identifying an LE/VSE-conforming component. The facility IDs assigned by IBM are:

CEE LE/VSE common library
EDC C language-specific library
IGZ COBOL language-specific library
IBM PL/I language-specific library

feedback code (fc). A condition token value. If you specify *fc* in a call to a callable service, a condition

token indicating whether the service completed successfully is returned to the calling routine.

file. A named collection of related data records that is stored and retrieved by an assigned name.

filename. A 1- to 7-character name used within an application and in JCL to identify a file. The filename provides the means for the logical file to be connected to the physical file.

fix-up and resume. The correction of a condition by changing the argument or parameter and running the routine again.

fixed-overflow. A condition raised as a result of an overflow during signed binary arithmetic or signed left-shift operations.

function. A routine that is invoked by coding its name in an expression. The routine passes a result back to the invoker through the routine name.

G

Gregorian calendar. The calendar in use since Friday, 15 October, 1582 throughout most of the world. Used as the basis for the *Lilian date* used in many LE/VSE date and time services.

H

handle cursor. Points to the first condition handler within the stack frame that is to be invoked when a condition occurs. As condition handling progresses, the handle cursor moves to earlier handlers within the stack frame, or to the first handler in the calling stack frame.

header file. A file that contains system-defined control information that precedes user data.

heap 0. Synonym for *initial heap*.

heap. An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments. See also *additional heap*, *anywhere heap*, *below heap*, *heap element*, and *initial heap*.

heap element. A contiguous area of storage allocated by a call to the CEEGTST service. Heap elements are always allocated within a single heap segment.

heap increment. See *increment*.

heap segment. A contiguous area of storage obtained directly from the operating system. The LE/VSE storage management scheme subdivides heap segments into individual heap elements. If the initial heap segment becomes full, LE/VSE obtains a second segment, or increment, from the operating system.

heap storage. See *heap*.

hexadecimal. A base 16 numbering system. Hexadecimal digits range from 0 through 9 (decimal 0 to nine) and uppercase or lowercase A through F (decimal ten to fifteen).

High Level Assembler. An IBM licensed program. Translates symbolic assembler language into binary machine language.

high level language (HLL). A programming language above the level of assembler language and below that of program generators and query languages.

HLL. High level language.

I

ILC. Interlanguage communication.

increment. The second and subsequent segments of storage allocated to the stack or heap.

indirect argument passing. The body of the argument list contains a pointer to the argument value.

indirect parameter passing. Placing an address in a parameter list. In other words, passing a pointer to a value instead of passing the value itself.

initial heap. The LE/VSE heap controlled by the HEAP run-time option and designated by a *heap_id* of 0. The initial heap contains dynamically allocated user data. See also *additional heap*.

initial heap segment. The first heap segment. A heap consists of the initial heap segment and zero or more additional segments or increments.

initial stack segment. The first stack segment. A stack consists of the initial stack segment and zero or more additional segments or increments.

initial program load (IPL). The process of loading system programs and preparing a system to run jobs.

IPL. Initial program load.

input procedure. A set of statements, to which control is given during the execution of a SORT statement, for the purpose of controlling the release of specified records to be sorted.

instance specific information (ISI). Located within the LE/VSE condition token, the ISI contains information used by the condition manager to identify and react to a specific occurrence of a condition.

integer. A positive or negative whole number or zero.

interactive. Pertaining to a program or system that alternately accepts input and responds. In an interactive system, a constant dialog exists between user and system. Contrast with *batch*.

interlanguage communication (ILC). The ability of routines written in different programming languages to communicate. ILC support allows the application writer to readily build applications from component routines written in a variety of languages.

interrupt. A suspension of a process, such as the execution of a computer program, caused by an event external to that process, and performed in such a way that the process can be resumed.

ISI. Instance specific information.

J

JCL. Job control language.

job control language (JCL). A sequence of commands used to identify a job to an operating system and to describe a job's requirements.

job step. One of a group of related programs complete with the JCL statements necessary for a particular run. Every job step is identified in the job stream by an EXEC statement under one job statement for the whole job.

Julian date. A date format that contains the year in positions 1 and 2, and the day in positions 3 through 5. The day is represented as 1 through 366, right-adjusted, with zeros in the unused high-order position.

K

KSDS. Key sequenced data sets. See *VSAM*.

L

Language Environment. A set of architectural constructs and interfaces that provides a common run-time environment and run-time services to applications compiled by Language Environment-conforming compilers.

Language Environment for VSE/ESA. An IBM software product that is the implementation of Language Environment on the VSE platform.

LE/VSE. Short form of Language Environment for VSE/ESA.

LE/VSE-conforming. Adhering to LE/VSE's common interface.

library. A collection of functions, subroutines, or other data.

library stack. An independent area of stack storage, allocated below the 16MB line, designed to be used only by library routines. See also *stack*, *user stack*, and *stack frame*.

library vector table (LIBVEC). A vector table used to support access to library routines (LE/VSE and HLLs) from compiler-generated code, user-written assembly language code, and other subroutines.

LIBVEC. Library vector table

LIFO. Last in, first out method of access. A queuing technique in which the next item to be retrieved is the item most recently placed in the queue.

Lilian date. The number of days since the beginning of the Gregorian calendar. Day one is Friday, 15 October 1582. The Lilian date format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

linked list. A list in which the data elements may be dispersed but in which each data element contains information for locating the next. Synonym for *chained list*.

linkage editor. A program that resolves cross-references between separately assembled object modules and then assigns final addresses to create a single relocatable phase. The linkage editor then stores the phase in a program library in main storage.

link-edit. To create a loadable computer program by means of a linkage editor.

local data. Data that is known only to the routine in which it is declared. Equivalent to local data in C and WORKING-STORAGE in COBOL.

locale. The definition of the subset of a user's environment that depends on language and cultural conventions.

locator. PL/I control block that holds the address of data such as structures or arrays and the address of the descriptor.

LWS. Library workspace.

M

main program. The first routine in an enclave to gain control from the invoker.

mapped condition. A condition that is generated by one component and converted, or mapped, to another component; for example, some LE/VSE conditions, such as the decimal divide condition that maps directly to the PL/I ZERODIVIDE condition.

megabyte (M). 1,048,576 bytes.

module. A language construct that consists of procedures or data declarations and can interact with other such constructs. In PL/I, an external procedure.

multitasking. See *multithreading*.

multithreading. Mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks, or threads.

N

n-way ILC application. An ILC application that includes a C routine, COBOL program, and PL/I routine.

NAB. Next available byte.

name scope. The portion of an application within which a particular declaration of external data applies or is known.

name space. The portion of a phase within which a particular declaration of external data applies or is known.

named heap. A heap set up specifically by the CEECRHP callable service. An identifier is returned when the heap is created.

national language support. Translation requirements affecting parts of licensed programs; for example, translation of message text and conversion of symbols specific to countries.

natural reentrancy. The attribute of applications that contain no static external data and do not require additional processing to make them reentrant. Contrast with *constructed reentrancy*.

nested condition. A condition that occurs during the handling of another, previous condition. LE/VSE by default permits 10 levels of nested conditions. You may change this setting by altering the DEPTHCONDLMT run-time option.

nested program. In COBOL, a program that is directly contained within another program.

nested enclave. A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also *child enclave* and *parent enclave*.

next available byte (NAB). The address of the next available byte of storage on a doubleword boundary. This address is a segment of stack storage.

next sequential instruction. The next instruction to be executed in the absence of any branch or transfer of control.

non-LE/VSE conforming. Any HLL program that does not adhere to LE/VSE's common interface. For example, VS COBOL II, DOS/VS COBOL, and DOS/VS PL/I are all non-LE/VSE conforming HLLs. Synonym for *pre-LE/VSE conforming*.

non-reentrant. A type of program that cannot be shared by multiple users.

O

object code. Output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

object deck. Synonym for *object module*.

object module. A portion of an object program suitable as input to a linkage editor. Synonym for *object deck*.

offset. The number of measuring units from an arbitrary starting point in a record, area, or control block, to some other point.

omitted parameter. A parameter not needed in a call.

online. Pertaining to a user's ability to interact with a computer.

ON-unit. The specified action to be taken upon detection of the condition named in the containing ON statement.

operating system. Software that controls the running of programs; in addition, an operating system may provide services such as resource allocation, scheduling, input/output control, and data management.

out-of-storage condition. A condition signaled when an application has used all of the storage allocated to it. If the STORAGE run-time option is set to a value other than 0, LE/VSE adds a reserve stack segment to the overflowing stack, and then signals the out-of-storage condition.

output procedure. A set of statements, to which control is given during the execution of a SORT statement after the sort function is completed, or during the MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow. That portion of an operation that exceeds the capacity of the intended unit of storage.

overlay. To write over existing data in storage.

owning stack frame. Given the calling sequence of Routine 1 calling Routine 2 that in turn calls Routine 3, Routine 3 is the owning stack frame if a condition occurs while Routine 3 is executing.

ON-unit. The specified action to be taken upon detection of the condition named in the containing ON statement.

P

packed decimal format. A format in which each byte in a field except the rightmost byte represents two numeric digits. The rightmost byte contains one digit and the sign. For example, the decimal value +123 is represented as 0001 0010 0011 1100.

pad. To fill unused positions in a field with dummy data, usually zeros, ones, or blanks.

parameter. Data items that are received by a routine.

parent enclave. The enclave that issues a call to system services or language constructs to create a nested (child) enclave. See also *child enclave* and *nested enclave*.

partition. A fixed-size division of storage.

Pascal. A high level language for general purpose use. Programs written in Pascal are block structured, consisting of independent routines.

pass by content. A COBOL argument passing style synonymous with passing an argument by value directly. In this style, R1 contains a pointer to a copy of the argument.

pass by reference. In programming languages, one of the basic argument passing semantics. The address of the object is passed. Any changes made by the callee to the argument value will be reflected in the calling routine at the time the change is made.

pass by value. In programming languages, one of the basic argument passing semantics. The value of the object is passed. Any changes made by the callee to the argument value will not be reflected in the calling routine.

percolate. The action taken by the condition manager when the returned value from a condition handler indicates that the handler could not handle the condition, and the condition will be transferred to the next handler.

phase. An application or routine in a form suitable for execution. The application or routine has been compiled and link-edited; that is, address constants have been resolved.

picture string. Character strings used to specify date and time formats.

PL/I. A general purpose scientific/business high level language. It is a high-powered procedure-oriented language especially well suited for solving complex scientific problems or running lengthy and complicated business transactions and record-keeping applications.

pointer. A data element that indicates the location of another data element.

portability. The ability to transfer an application from one platform to another with relatively few changes to the source code.

PPA1 entry point block. Program Prolog Area. This block contains information about the compiled module.

PPA2 entry point block. An extension of the PPA1 entry point block.

PPT. Processing Program Table

precedence. In programming languages, an order relation defining the sequence of the application of operations or options.

precision. A measure of the ability to distinguish between nearly equal values. See also *single-precision* and *double-precision*.

pre-initialization. A facility that allows a routine to initialize the run-time environment once, perform multiple executions within the environment, then explicitly terminate the environment.

pre-LE/VSE conforming. Any HLL program that does not adhere to LE/VSE's common interface. For example, VS COBOL II, DOS/VS COBOL, and DOS/VS PL/I are all pre-LE/VSE conforming HLLs. Synonym for *non-LE/VSE conforming*.

preprocessor. A routine that examines application source code for preprocessor statements that are then executed, resulting in the alteration of the source.

procedure. A named block of code that can be invoked, usually via a call. In LE/VSE, the term *routine* is used as generic for a procedure or a function.

process. The highest level of the LE/VSE program management model. A process is a collection of resources, both program code and data, and consists of at least one enclave.

Processing Program Table (PPT). A CICS table that contains information about CICS phases (whether the phase is in storage or not, its language, use count and entry point address, etc.) needed to complete a transaction.

program. See *application program*.

program control data. In PL/I, data used to affect how a program runs; that is, any data that is not string or arithmetic data.

program management. The functions within the system that provide for establishing the necessary

activation and invocation for a program to run in the applicable run-time environment when it is called.

program mask. A structure that describes the manner in which S/370 hardware-detected conditions are to be handled.

program status word (PSW). An area in storage used to indicate the order in which instructions are executed and to hold and indicate the status of the operating system. The program mask (bits 20 to 23) of the PSW can be manipulated to enable or disable the detection of some hardware conditions under LE/VSE.

program unit. Synonym for *compilation unit*.

programmable workstation (PWS). A workstation that has some degree of processing capability and that allows a user to change its functions.

program unit. Synonym for *compilation unit*.

prolog. The code sequence when a routine is entered.

promote. To change a condition. A condition is promoted when a condition handling routine changes the condition to a different one. A condition handling routine promotes a condition because the error needs to be handled in a way other than that suggested by the original condition.

PSB. Program specification block.

PSW. Program status word.

Q

q_data. Qualifying data. Information that a user-written condition handler can use to identify and react to a given instance of a condition.

q_data_token. An optional 32-bit data object that is placed in the ISI. It is used to access math condition qualifying data associated with a given instance of a condition.

R

reason code. A value returned to the invoker of an enclave that indicates how the enclave terminated. The value reflects whether the enclave terminated successfully, or unsuccessfully, to an unhandled condition.

recursive routine. A routine that can call itself or be called by another routine that it has called.

reentrant. The attribute of a routine or application that allows more than one user to share a single copy of a phase.

register. (1) Special processing areas that hold a specific amount of data and can process, load, and store this data quickly. (2) To specify formally. In LE/VSE, to register a condition handler means to add a user-written condition handler onto a routine's stack frame.

register save area (RSA). Area of main storage in which contents of registers are saved.

resident routines. A category of LE/VSE library routines linked with your application. They include such things as initialization routines and *callable service stubs*.

resume. To begin execution in an application at the point immediately after which a condition occurred. A resume occurs when the condition manager determines that a condition has been handled and normal application execution should continue.

resume cursor. Designates the point in the application where a condition occurred when it is first reported to the condition manager. The resume cursor also designates the point where execution resumes after a condition is handled, usually at the instruction in the application immediately following the point at which the error occurred. The resume cursor can be moved with the CEEMRCR callable service.

return code. A code produced by a routine to indicate its success. It may be used to influence the execution of succeeding instructions.

return_code_modifier. A value set by LE/VSE routines that manage the environment. It indicates whether or not an enclave terminated successfully.

rollback. The backing out of any updates made by a failing application.

root phase. The phase containing a main routine and the first to be executed in an application.

routine. In this book, used as an exact equivalent of a COBOL/VSE *compilation unit*, and means a named external routine, with or without named entry points, and with or without internal (contained) routines.

RMODE. Residence mode. The attribute of a phase that specifies whether the phase, when loaded, must reside below the 16MB virtual storage line or may reside anywhere in virtual storage.

RRDS. Relative record data sets. See *VSAM*.

run. To cause a program, utility, or other machine function to be performed.

run time. Any instant at which a program is being executed. Synonym for *execution time*.

run-time environment. A set of resources that are used to support the execution of a program. Synonym for *execution environment*.

run unit. One or more object programs that are executed together. In LE/VSE, a run unit is the equivalent of an *enclave*.

S

safe condition. Any condition having a severity of 0 or 1. Such conditions are ignored if no condition handler handles the condition.

save area. Area of main storage in which contents of registers are saved.

scalar. A quantity characterized by a single value. Contrast with *aggregate*.

scope. 1. A term used to describe the effective range of the enablement of a condition and/or the establishment of a user-generated routine to handle a condition. Scope can be both statically and dynamically defined. 2. The portion of an application within which the definition of a variable remains unchanged.

segment. See *stack segment*.

severity code. A part of run-time messages that indicates the severity of the error condition (1, 2, 3, or 4).

shared virtual area (SVA). In VSE, an area of main storage containing a system directory list (SDL) of frequently used phases, resident routines shared between partitions, and an area for system support. The presence of a reentrable routine in the SVA saves loading time when the routine is needed.

significance condition. A condition raised when the resulting fraction in a floating-point addition or subtraction operation is zero.

single-precision. Pertaining to the use of one computer word to represent a number in accordance with the required precision. Needed for proper alignment. See also *precision* and *double-precision*.

sort/merge program. A processing program that can be used to sort or merge records in a prescribed sequence.

source code. The input to a compiler or assembler, written in a source language.

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run.

stack. An area of storage used for suballocation of stack frames. Such suballocations are allocated and freed on a LIFO (last in, first out) basis. A stack is a collection of one or more stack segments consisting of an initial stack segment and zero or more increments.

stack frame. The physical representation of the activation of a routine. The stack frame is allocated on a LIFO stack and contains various pieces of information including a save area, condition handling routines, fields to assist the acquisition of a stack frame from the stack, and the local, automatic variables for the routine. In LE/VSE, a stack frame is synonymous with *DSA*.

stack frame collapse. An action that occurs when the condition manager skips over one or more active routines and execution resumes in an earlier routine on the stack. A stack frame collapse happens if an explicit GOTO is coded in a PL/I routine or if the resume cursor is moved with the CEEMRCR.

stack segment. A contiguous area of storage obtained directly from the operating system. The LE/VSE storage management scheme subdivides stack segments into individual DSAs. If the initial stack segment becomes full, a second segment or increment is obtained from the operating system.

stack storage. See *stack* and *automatic storage*.

standard system action. The name given to the language-defined default action taken when a condition occurs and it is not handled by a condition handler.

statement. In programming languages, a language construct that represents a step in a sequence of actions or a set of declarations.

static call. A call that results in the resolution of the called program statically at link-edit time. Contrast with *dynamic call*.

static data. Data that retains its last-used state across calls.

static storage. Storage that persists and retains its value across calls. Contrast with *dynamic storage*.

storage heap. An unordered group of program stack areas that may be associated with programs running within a process.

suboption. An option that can be used with compile-time and run-time options to further specify the action of the option.

subroutine. In general, any routine within an application called by another routine.

symbolic feedback code. The symbolic representation of the 12-byte condition token returned by LE/VSE callable services. Symbolic feedback codes are provided so that you do not have to code the entire 12-byte condition token in a condition handling routine.

system directory list (SDL). In VSE, a list containing the directory entries of frequently-used phases and of all phases resident in the SVA. The list resides in the SVA.

subsystem. A secondary or subordinate system, or programming support, usually capable of operating independently of or asynchronously with a controlling system. Example: CICS.

SVC. Supervisor call. A request that serves as the interface to certain functions, such as the allocation of storage.

syntax. The rules governing the structure of a programming language and the construction of a statement in a programming language.

T

thread. The basic run-time path within the LE/VSE program management model. It is dispatched by the system with its own instruction counter and registers. The thread is where actual code resides.

token. See *condition token*.

translator. See *CICS translator*.

transient data queue. A file to which run-time messages are written under CICS. Under LE/VSE, the name of this file is CESE.

termination imminent step. The final step of the 3-step LE/VSE condition handling model. In the termination imminent step, user-written condition handlers and PL/I ON-units are given one last chance to handle a condition or perform cleanup before the thread is terminated. See also *condition step* and *enablement step*.

U

underflow condition. A condition that occurs when the result characteristic of a floating-point operation is less that zero and the result fraction is not zero. In an extended-format floating-point result, the condition is raised only when the high-order characteristic overflows.

unpacked decimal format. A format for representing numbers in which the digit is contained in bits 4 through 7 and the sign is contained in bits 0 through 3 of the rightmost byte. Bits 0 through 3 of all other bytes contain 1s (hex F). For example, the decimal value of +123 is represented as 1111 0001 1111 0010 1111 0011. Synonym for *zoned decimal format*.

user-written condition handler. A routine established by the CEEHDLR callable service to handle a condition or conditions when they occur in the common run-time environment. A queue of user-written condition handlers established by CEEHDLR may be associated with each stack frame in which they are established.

user exit. A routine that takes control at a specific point in an application. Two assembler user exits and one HLL user exit are provided by LE/VSE. They are invoked to perform initialization functions and both normal and abnormal termination functions.

user heap. See *initial heap*.

user stack. An independent area of stack storage that may be located above or below 16MB, designed to be used by both library routines and compiled code. See also *stack*, *stack frame*, *library stack*.

V

vendor. A person or company that provides a service or product to another person or company.

VSTRING. The VSTRING data type is used for the character string parameters in many of the LE/VSE callable services. In Language Environment/VSE Version 1 Release 4, VSTRING is a halfword length-prefixed character string for input, or a fixed-length 80-character string for output.

VSE (Virtual Storage Extended). A system that consists of a basic operating system (VSE/Advanced Functions) and IBM-supplied programs required to meet the data processing needs of a user.

W

weak external reference. A special type of external reference that is not to be resolved by automatic library calls unless an ordinary external reference to the same symbol is found. The external symbol dictionary entry specifies the symbol; the location is unknown.

word. A contiguous series of 32 bits (4 bytes) in storage, addressable as a unit. The address of the first byte of a word is evenly divisible by four.

working storage. In COBOL/VSE, the storage required for data items in the WORKING-STORAGE SECTION. Working storage is a portion of main storage that is used by a computer program to hold data temporarily.

Z

zoned decimal format. Synonym for *unpacked decimal format*.

Index

Special characters

, (comma) 38
/ (slash)
specifying in parameter list 22, 38
@DELETE service routine for preinitialization
components of 384
return/reason codes for 384
@FREESTORE service routine for preinitialization
components of 385
return/reason codes for 386
@GETSTORE service routine for preinitialization
components of 384
return/reason codes for 385
@LOAD service routine for preinitialization
components of 383
return/reason codes for 383, 384
@MSGRTN service routine for preinitialization
components of 386
return/reason codes for 386
& (ampersand) 44
__csplist macro 405
__osplist macro 404
__pcblist macro 405
__R1 macro 404
#pragma directives
See pragma
abort() function
C condition handling semantics and 124
HLL user exit and 332
in a preinitialized environment 367
SIGABRT and 122
argc parameter for C
C parameter passing styles and 405
argv parameter for C
C parameter passing styles and 405
atexit list
CEEPIPI and 366
calloc() function 85
ctdli()interface to DL/I 313
exit() function 124
C condition handling scenario and 124
CEEBINT HLL user exit and 332
in a preinitialized environment 367
fetch()
C fetching C 309
fprintf function 201
printf() function
default destination 201
interspersing messages into an application 200
raise() function for C
how C terminology differs from LE/VSE's 123
raise() function for C *(continued)*
SIGABRT
See abort() function
SIGTERM
HLL user exit and 332
return() 367
stderr
default destinations of 200
' (apostrophe) 44

A

abend
CANCEL system macro and 359
CEEPIPI interface to preinitialization 376
CICS
assembler user exit and EXEC CICS ABEND 326
EXEC CICS HANDLE ABEND and 306
forcing database rollback 308
nested conditions and 394
short-on-storage condition and 304
codes
abend AKCP (CICS short-on storage condition) 304
CEEAUE_RETURN field of CXIT control block and 324
exempting from condition handling with ABPERC run-time option 108
exempting from condition handling with assembler user exit 108
in CICS 307
DB2 312
definition 106
DL/I 315
dump, requesting 326
exempting from condition handling
ABPERC run-time option and 108
CEEBXITA and 108, 323
LE/VSE-generated 108
nested conditions and 140
nested enclaves and 394
q_data_token and 182
requesting
using CEE5ABD 322
short-on-storage condition can cause 304
sort and merge operations 410
system
DB2 and CEEBXITA 312
DL/I and CEEBXITA 315
TRAP run-time option and 323
abnormal termination
See abend
ABPERC run-time option
description 33
ABTERMENC run-time option 312, 315
abend codes and 71
description 33
using to terminate with abend code or return and reason codes 70
ACTION linkage editor control statement 24, 25
add_entry
CEEPIPI(init_main) and 380
CEEPIPI(init_sub) and 380
return codes from 381
syntax description 380
additional heap
intended purpose of 85, 87
tuning the heap 87
addressing mode
See AMODE
AIXBLD run-time option
description 33
ALL31 run-time option
description 33
ALLOCATE
statement for PL/I 85
AMODE 87
ALL31 run-time option and 89
assembler routines and 336
C AMODE considerations 9, 300
for CEEBXITA user exit 323
heap storage 87
in preinitialized routines 364
ampersand (&) 44
ANYHEAP run-time option
description 33
anywhere heap 85, 87
apostrophe (') 44
application
See enclave
application defaults 36
AREA storage for PL/I 85
ARGPARSE run-time option
description 33
argument
distinguishing program arguments from run-time options 38
list format
EXECOPS run-time option and 405, 406
how interactions of EXECOPS and PLIST run-time options affect 405
PLIST run-time option and 405
passing
by reference 50, 51
by value 50, 51
C style 51, 403
COBOL styles 51
directly 50, 51
indirectly 50, 51
PL/I styles 51
styles permitted by LE/VSE 51, 401

argument *(continued)*
relationship to parameter list 50
specifying to an invoked routine which format to expect (C) 405
arithmetic
See also math services
date calculations
examples illustrating date and time callable service 222, 237
overview 216
ASMTDLI interface to DL/I 313
ASSEMBLE file 185
assembler language
ASMTDLI interface 313
COBOL parameter list format 406
macros 341, 342
CEECAA—generate a CAA mapping 344
CEECIB— generate a CIB mapping 345
CEEDSA—generate a DSA mapping 345
CEEENTRY—generate an LE/VSE-conforming prolog 341
CEEFETCH— dynamically load an LE/VSE routine that can be later deleted 350
CEELOAD— dynamically load an LE/VSE routine 348
CEEPPA—generate a PPA 345
CEERELES— dynamically delete an LE/VSE routine 353
CEETERM—terminate an LE/VSE-conforming routine 344
routines
calling conventions for 335
compatibility with LE/VSE 335
condition handling for 165, 170, 336
equivalent callable services for 359
examples 355, 359
invoking callable services from 359
main routines 335, 355, 356
no support for assembler main routines under CICS 335
operating services for 359, 401
program check, handling 165
subroutines 336, 356, 358
user exit
See CEEBXITA assembler user exit
assign
message insert data 196
ATTACH system macro 359
automatic data
definition 76
how used in enclave 78
automatic library lookup (AUTOLINK) feature
See librarian automatic library lookup (AUTOLINK) function

B

below heap
what used for 87
BELOWHEAP run-time option
description 33
BYADDR compile-time option for PL/I 55
BYVALUE compile-time option for PL/I
functions 55
required if SYSTEM(CICS) specified 407

C

C
`#pragmas`
See pragma
`stderr`
default destinations of 200
interleaving output with other output 201
redirecting output from 201
AMODE/RMODE considerations 9
building linked list in 90, 91
calls to C under CICS 309
condition handling 121, 127
examples
CEE5CTY, CEEFMDT, and CEEDATM 249, 251
CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST and CEEDSHP 96, 97
CEEDAYS, CEEDATE and CEEDYWK 238, 239
CEEGTST and CEEFRST 90, 91
CEEHDLR, CEEGTST, CEECZST and CEEMRCR 149, 152
CEEHDLR, CEESGL, CEEGQDT and CEEMRCR 158, 160
CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and CEEMSG 205, 207
CEEQCEN and CEESCEN 219
CEESECS and CEEDATM 226, 227
CEESECS, CEESECI, CEEISEC, and CEEDATM 231, 233
CEESECS, multiple calls to 222
CEESSLOG 290
coding main routine to receive inbound parameter list 52, 55
global condition handling model 121
interfaces to DL/I from 313
L-names
See L-names
LONGNAME compile-time option 11
parameter passing style 51
PLIST and EXECOPS interactions 405
styles 403, 405
pragma
See pragma
prelinker
See prelinker
RENT compile-time option
making C routines reentrant with 58
S-names
See S-names
C *(continued)*
signals
mapping abends to 128
mapping S/370 exceptions to 127
specifying run-time options for 36
CAA (common anchor area)
See common anchor area (CAA)
CALL statement
for COBOL
callable service feedback code and 175
for PL/I
See FETCH statement
call_main
See preinitialization facility, CEEPIPI(call_main)
call_sub
See preinitialization facility, CEEPIPI(call_sub)
call_sub_addr
See preinitialization facility, CEEPIPI(call_sub_addr)
callable services
CEE5ABD—terminate enclave with an abend 64
CEE5CIB—return pointer to condition information block 104
CEE5CTY—set default country 247
CEE5DMP—generate dump 281
CEE5GRC—get the enclave return code 64
CEE5GRN—get name of routine that incurred condition 104
CEE5GRO— get offset of instruction 104
CEE5LNG—set national language 247
CEE5MCS—obtain default currency symbol 247
CEE5MDS—obtain default decimal separator 247
CEE5MTS—obtain default thousands separator 247
CEE5PRM—query parameter string 64, 281
CEE5PRML 281
CEE5RPH—set report heading 82
CEE5SPM—query and modify LE/VSE hardware condition enablement 104
CEE5SRC—set the enclave return code 64
CEE5SRP— set position for execution to resume 104
CEE5TSTG 281
CEE5USR—set or query user area fields 281
CEECBLDY—convert date to COBOL Lilian format 214
CEECMI—store and load message insert data 185
CEECRHP—create new additional heap 82
CEECZST—reallocate (change size of) storage 82
CEEDATE—convert Lilian date to character format 214

callable services *(continued)*
CEEDATM—convert seconds to character timestamp 214
CEEDAYS—convert date to Lilian format 214
CEEDCOD—decompose a condition token 171
CEEDSHP—discard heap 82
CEEDYWK—calculate day of week from Lilian date 214
CEEFMDA—obtain default date format 247
CEEFMDT—obtain default date and time format 247
CEEFMON—format monetary string 257
CEEFMTM—obtain default time format 247
CEEFRST—free heap storage 82
CEEFTDS—format date and time into character string 257
CEEGMT—get current Greenwich mean time 214
CEEGMTO—get offset from Greenwich mean time to local time 214
CEEGPID—retrieve LE/VSE version and platform ID 281
CEEGQDT—retrieve q_data_token 104, 158, 160
CEEGTST—get heap storage 82
CEEHDLR—register user condition handler 104
CEEHDLU—unregister user condition handler 104
CEEISEC—convert integers to seconds 214
CEEITOK—return initial condition token 104
CEELCNV—query locale numeric conventions 257
CEELOCT—get current local time 214
CEEMGET—get a message 185
CEEMOUT—dispatch a message 185
CEEMRCE— move resume cursor to an explicit location 105
CEEMRCR—move resume cursor relative to handle cursor 104
CEEMSG—get, format, and dispatch a message 185
CEENCOD—construct a condition token 171
CEEQCEN—query the century window 214
CEEQDTC—query locale, date, and time conventions 257
CEEQRYL—query active locale environment 257
CEERAN0—calculate uniform random numbers 281
CEESCEN—set the century window 214
CEESCOL—compare string collation weight 257
CEESECI—convert seconds to integers 214

callable services *(continued)*
CEESECS—convert timestamp to number of seconds 214
CEESETL—set locale operating environment 257
CEESGL—signal a condition 104
CEESTXF—transform string into collation weights 257
CEETEST—invoke debug tool 281
feedback code parameter
See feedback code
getting started with 283, 286
guidelines for writing 401, 403
invoking 359
assembler 359

calls
dynamic call
See also CALL statement
calls between COBOL/VSE and VS COBOL II, under CICS 309
external references resolved at run time when made 7
static call
external references resolved at link-edit time when made 7
in CICS COBOL applications 309, 310

cancel codes
See abend, codes

CANCEL system macro
CEE5ABD callable service and 359
CEESGL callable service and 359
table of equivalent LE/VSE services 359

casting, when using R1 and osplist macros 404

CBLOPTS run-time option
description 33
VS COBOL II compatibility and 39

CBLPSHPOP run-time option
description 33
EXEC CICS PUSH and EXEC CICS POP commands and 307

CBLTDLI interface to DL/I 313

CCE5MCS callable service
Euro support 249

CDLOAD and CDDELETE system macros 363

CEE facility ID 198

CEE5ABD—terminate enclave with an abend
CANCEL system macro and 359

CEE5CIB—return pointer to condition information block 104

CEE5CTY—set default country
examples using
examples with CEEFMDT and CEEDATM 249, 255

CEE5DMP—generate dump
CESE transient data queue and 309
description 281
PDUMP system macro and 363

CEE5GRN—get name of routine that incurred condition
examples using 160, 164

CEE5GRO— get offset of instruction 104

CEE5LNG—set national language messages and 199

CEE5PRM—query parameter string
description 281

CEE5PRML 64

CEE5PRML—pass string with length 300 characters 281

CEE5SPM—query and modify LE/VSE hardware condition enablement
advisory note regarding 401
condition handling, XUFLOW run-time option and 108
examples using 160, 164

CEE5SRP— set position for execution to resume 104

CEE5TSTG—test for access available to a storage address 281

CEE5USR—set or query user area fields
description 281

CEEAUE_ABND field of CXIT control block 326

CEEAUE_ABTERM field of CXIT control block 325

CEEAUE_CODES field of CXIT control block
description 327
specifying abend codes in 323

CEEAUE_DUMP field of CXIT control block 326

CEEAUE_FBCODE field of CXIT control block 328

CEEAUE_FLAGS field of CXIT control block
CEEAUE_ABND field of 326
CEEAUE_ABTERM field of 325
CEEAUE_DUMP field of 326
format 325

CEEAUE_FUNC field of CXIT control block 324

CEEAUE_LEN field of CXIT control block 324

CEEAUE_OPTION field of CXIT control block 327

CEEAUE_PARM field of CXIT control block 326

CEEAUE_REASON field of CXIT control block
description 325
relationship to CEEAUE_ABND 326
relationship to CEEAUE_RETURN 326

CEEAUE_RETURN field of CXIT control block
description 324
relationship to CEEAUE_ABND 324, 326
relationship to CEEAUE_REASON 325, 326

CEEAUE_USER field of CXIT control block 327

CEEAUE_WORK field of CXIT control block 327

CEEBINT HLL user exit 64
functions 320
interactions with CEEPIPI 366
interface to 332
languages it can be coded in 332

CEEBINT HLL user exit *(continued)*
terminating enclave using 332
user word parameter of, and CEEAUE_USER 333
when invoked 321
CEEBLDTX EXEC
error messages 192
using to create message files 185
CEEBXITA assembler user exit 64
abends and
requesting 322
specifying codes to be exempted from condition handling 323
actions taken if errors occur within the exit 323
AMODE/RMODE considerations 323
application-specific 319
behavior of
during enclave initialization 320, 321, 322
during enclave termination 322
during process termination 322
DB2 and 312
DL/I and 315
EXEC CICS commands that cannot be used with 307
functions 319
installation-wide 319
interactions with CEEPIPI 366
modifications to, rules for making 323
PLIRETC and 307
specifying run-time options in 300, 311, 314, 327
TRAP run-time option and 323
when invoked 321
work area for 327
CEECAA assembler macro
relationship to CEEENTRY 341
syntax description 344
CEECBLDY—convert date to COBOL Lilian format
example using 245
CEECIB assembler macro 345
CEECMI—store and load message insert data
assigning values to message insert 196
CEECOPT assembler source file 36
CEECOPT options module
CEEXOPT macro and 40
description 36
IBM-supplied version of defaults 40
specifying run-time options with 300
CEECXITA assembler user exit 308
See CEEBXITA assembler user exit
CEEDATE—convert Lilian date to character format
examples using 238
CEEDATM—convert seconds to character timestamp
examples using
examples with CEE5CTY and CEEFMDT 249, 255
examples with CEESECS 226, 231
CEEDATM—convert seconds to character timestamp *(continued)*
examples using *(continued)*
examples with CEESECS, CEESECI, and CEEISEC 231
CEEDAYS—convert date to Lilian format
examples using 238
CEEDCOD—decompose a condition token
examples using 205
testing equivalent tokens 175
CEEDOPT assembler source file 35
CEEDOPT options module
CEEXOPT macro and 40
description 35
IBM-supplied version of defaults 40
specifying run-time options with 30, 311, 314
CEEDSA assembler macro
relationship to CEEENTRY 341
syntax description 345
CEEDSASZ label 345
CEEDYWK—calculate day of week from Lilian date
examples using 238
CEEENTRY assembler macro
relationship to CEECAA 341
relationship to CEEDSA 342
relationship to CEEPPA 341, 342
relationship to CEETERM 341
syntax description 341
CEEFETCH assembler macro 350
CEEFMDA—obtain default date format 247
CEEFMDT—obtain default date and time format
examples using
examples with CEE5CTY and CEEDATM 249, 255
CEEFMON—format monetary string
examples using 258, 260
CEEFMTM—obtain default time format 247
CEEFTDS—format date and time into character string
examples using 261, 263
CEEGMT—get current Greenwich mean time 214
CEEGMTO—get offset from Greenwich mean time to local time 214
CEEGPID—retrieve the LE/VSE version and platform ID
description 281, 282
CEEGQDT—retrieve q_data_token
examples using 158, 160
CEEHDLR—register user condition handler 115
assembler routines and 336
condition handling example 127
condition handling model and 115
condition handling terminology 122
examples using
assembler example 165, 170
examples with CEE5SPM, CEE5GRN and CEEMOUT 160, 164
CEEHDLR—register user condition handler *(continued)*
examples using *(continued)*
examples with CEEGTST, CEECZST and CEEMRCR 149, 157
examples with CEESGL, CEEGQDT and CEEMRCR 158, 160
restrictions on using with various EXEC CICS commands 306, 359
SETRP command and 359
STXIT system macro and 359
syntax description of user-written condition handlers 138, 139
CEEHDLU—unregister user condition handler
EXEC CICS HANDLE ABEND command and 359
SETRP command and 359
STXIT system macro and 359
syntax description of user-written condition handlers 138, 139
CEEISEC—convert integers to seconds
examples using 231
CEELCNV—query locale numeric conventions
examples using 264, 266
CEELOAD assembler macro 348
CEELOCT—get current local time
examples using 216
CEELRR—initialize or terminate library routine retention 339
CEEMGET—get a message
examples using 205
examples with CEEMOUT, CEENCOD, CEEDCOD and CEEMSG 207, 211
relationship to condition tokens and other message services 174
CEEMOUT—dispatch a message
examples using
examples with CEEHDLR, CEE5SPM and CEE5GRN 160, 164
examples with CEENCOD, CEEMGET, CEEDCOD and CEEMSG 205, 207, 211
relationship to condition tokens and other message services 174
WTO system macro and 363
CEEMRCE— move resume cursor to an explicit location 105
CEEMRCR—move resume cursor relative to handle cursor
examples using
examples with CEEHDLR, CEEGTST and CEECZST 149, 157
examples with CEEHDLR, CEESGL and CEEGQDT 158, 160
resume action and 116
CEEMSG—get, format, and dispatch a message
examples using 205

CEEMSG—get, format, and dispatch a message *(continued)*
relationship to condition tokens and other message services 174
CEENCOD—construct a condition token
examples using 205
CEEPIPI
See preinitialization facility
CEEPIPI(add_entry) 380
CEEPIPI(call_main) 373
CEEPIPI(call_sub_addr) 376
CEEPIPI(call_sub) 374
CEEPIPI(delete_entry) 381
CEEPIPI(end_seq) 378
CEEPIPI(init_main) 369
CEEPIPI(init_sub_dp) 371
CEEPIPI(init_sub) 370
CEEPIPI(start_seq) 377
CEEPIPI(term) 379
CEEPPA assembler macro
relationship to CEEENTRY 341
syntax description 345
CEEQCEN—query the century window
examples using 219, 221
CEEQDTC—query locale, date, and time conventions
examples using 267, 269
CEEQRYL—query active locale environment
examples using 273, 278
CEERAN0—calculate uniform random numbers
description 281
CEERELES assembler macro 353
CEESCEN—set the century window
examples using 219, 221
CEESCOL—compare string collation weight
examples using 270, 272
CEESECI—convert seconds to integers
examples using 231
CEESECS—convert timestamp to number of seconds
examples using CEESECS
C 222
COBOL 223
PL/I 225
examples with CEEDATM
C 226, 227
COBOL 228, 229
PL/I 230, 231
examples with CEESECI, CEEISEC, and CEEDATM
C 231, 233
COBOL 233, 235
PL/I 236
CEESETL—set locale operating environment
examples using 264, 269, 273, 275
CEESGL—signal a condition
CANCEL system macro and 359
description of signals 106
examples using 158, 160
EXEC CICS HANDLE ABEND command and 359
HLL-specific condition handlers and 106, 108
CEESGL—signal a condition *(continued)*
important condition handling terminology 116
relationship to condition tokens and message services 174
SETRP command and 359
STXIT system macro and 359
TRAP run-time option does not affect 108
user-written condition handlers and 108
CEESTART
default entry point for PL/I 6
CEESTXF—transform string into collation weights
examples using 276, 278
CEETDLI interface to DL/I 313
CEETERM assembler macro
relationship to CEEENTRY 341
syntax description 344
CEETEST—invoke debug tool
condition handling and 114
description 281
CEEUOPT assembler source file 36
CEEUOPT options module
CEEXOPT macro and 40
description 36
specifying run-time options with 30, 300, 311, 314
CEEXOPT macro
description 40
sample of CEECOPT modified using 42
sample of CEEDOPT modified using 41
sample of CEEUOPT modified using 43
usage notes for 44, 45
CEEXPIT macro 365
CEEXPITS macro 365
CEEXPITY macro 365
CESE transient data queue
CICS dump and message output file 303, 308
format 308
message handling and 200
CHAP command 359
CHECK run-time option
description 33
CICS
See also EXEC CICS command
callable service behavior under
availability of callable services 303
CBLPSHPOP run-time option and 307
CEECOPT options module 36
CESE transient data queue and 303
CICS partition 297
CICS run unit
behavior in nested enclave 394
compared to LE/VSE enclave 297
COBOL parameter list formats 406
coding main routines to receive parameters 54
condition handling for 305, 308
DL/I interface 313
CICS *(continued)*
DOS/VS COBOL compatibility considerations 303
I/O restrictions in 298
link-editing for 299
message and dump output file 303
message format 308
message handling for 308
multi-tasking for 298
PLIRETC support 299, 307
PLIRETV support 299
PLIST and EXECOPS interactions 406
processing program table (PPT) 298
program control table (PCT) 298
reentrancy and 57
required level of 297
run-time option behavior under 301, 302
run-time output file 308
sort parameter list and 413
specifying run-time options for 300
storage and 303
SYSTEM(CICS) compile-time option and 54, 56, 406
terminology 297
transaction 297, 298
transaction rollback 307
translator 298, 306
CICS-wide run-time options, printing to console 45
CLOSE system macro 363
COBOL
building a linked list in 92, 93
can choose between static and dynamic calls under 7
CEEBXITA assembler user exit for VS COBOL II compatibility 320
condition handling 128, 132
constructing and dispatching a message for the significance condition 160, 164
DOS/VS COBOL under CICS 303
examples
CEE5CTY, CEEFMDT, and CEEDATM 251, 253
CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST and CEEDSHP 98, 99
CEECBLDY—convert date to COBOL Lilian format 245
CEEDAYS, CEEDATE and CEEDYWK 240, 242
CEEFMON—format monetary string 259
CEEFTDS—format date and time into character string 261
CEEGTST and CEEFRST 92, 93
CEEHDLR, CEE5SPM, CEE5GRN and CEEMOUT 160, 164
CEEHDLR, CEEGTST, CEECZST and CEEMRCR 153, 157
CEELCNV and CEESETL 264
CEEMOUT, CEENCOD, CEEMGET, CEEDCOD and CEEMSG 207, 209

COBOL *(continued)*
examples *(continued)*
CEEQCEN and CEESCEN 220, 221
CEEQDTC and CEESETL 267
CEESCOL—compare string collation weight 270
CEESECS and CEEDATM 228, 229
CEESECS, CEESECI, CEEISEC and CEEDATM 233, 235
CEESECS, multiple calls to 223
CEESETL and CEEQRYL 273
CEESSLOG 291
CEESTXF and CEEQRYL 276
coding main routine to receive inbound parameters 52, 55
GOBACK statement
generates return code 69
interfaces to DL/I from 313
non-CICS DOS/VS COBOL programs supported in single enclave only 393
order of program arguments and run-time options 39
parameter list formats 406
parameter passing style 51
RENT compile-time option for COBOL 58
specifying run-time options from 35, 45
STOP RUN statement
preinitialized enviroment and 367
return codes and 69
comma (,) 38
COMMAREA
COBOL user-written condition handlers and 306
common anchor area (CAA)
writing assembler routines 336
common area in linkage editor map 27
common run-time environment, introduction 3
compatibility
assembler 335
CICS 300
condition
callable service feedback code and 173, 175
definition 106
divide-by-zero
examples illustrating condition handling for 158
nested 394
severity
CEEBXITA assembler user exit and 325
COBOL condition handling 129
condition token and 172
ERRCOUNT run-time option and 110
how to determine in a message 111, 198
TERMTHDACT run-time option and 113
unhandled conditions and 70
condition *(continued)*
signalling with CEESGL callable service
See CEESGL—signal a condition
condition handler
C signal handlers
CEESGL callable service and 108
description 123
TRAP run-time option and 108
description 115
HLL semantics
percolation and 116
SORT and MERGE operations 410
TRAP run-time option and 108
PL/I ON-units
CEESGL callable service and 108
SORT and MERGE operations 412
TRAP run-time option and 108
user-written
accessing a q_data structure and moving the resume cursor from 158, 160
C `raise()` function and 122, 123
C `signal()` function and 123
CEESGL callable service and 108
coding 137, 139
constructing message string when significance condition occurs 160, 164
EXEC CICS commands that cannot be used with 306
in ILC applications 140
in nested condition handling 140
introduction to user-written condition handlers 114
registering with CEEHDLR callable service 115
role in LE/VSE condition handling model of 114
sort and merge operations and 410, 412
syntax for 138
TRAP run-time option and 108
condition handling 173
See also stack, frame
assembler routines 336
basic condition handling scenarios 116, 121
C semantics 121, 127
`signal()` function and 122, 123
default actions for C conditions 121
example of 125, 127
global error table and 121
scenario of 124
callable service feedback code and 173, 175
callable services for
examples using CEEHDLR, CEEGTST, CEECZST and CEEMRCR 149, 157
examples using CEEHDLR, CEESGL, CEEGQDT and CEEMRCR 158, 160
usage scenario 148
condition handling *(continued)*
CICS, under 306
COBOL
ON SIZE ERROR clause 128, 132
semantics of 128, 132
coding 137, 139
condition step of
See condition step
default actions 129
enablement step of
See enablement, condition handling step
examples 141, 170
global model provided by C 121
introduction to 103, 116
nested enclaves
created by C`system()` function 395
created by EXEC CICS LINK or EXEC CICS XCTL 394, 395
PL/I 132, 136
registering with USRHDLR 139
signalling condition with CEESGL
See CEESGL—signal a condition
sort and merge considerations 410
COBOL/VSE 410
PL/I 412
stack frame-based model provided by LE/VSE
details of 116
overview 103, 104, 116
termination imminent step of
See termination imminent step
terminology 106
user exits and 307
user-written condition handler 114
using symbolic feedback code in 175, 181
when to use 103
condition manager
C signal handler and 127
percolating abends and 129
stack frame collapse and 131
symbolic feedback code and 176
thread initialization and 65
condition step 109, 111
condition token 171, 185
C signal and 182
callable service feedback code and 173, 175
condition handling model and 103
messages and 199
constructed reentrancy 57
See prelinker
continuations 44
control
block
CAA 336
sections 26
COPY file 185
COUNTRY run-time option
description 33
critical error message (severity 4) 198
cross system product (CSP) 405
csplist macro 405
CXIT control block
CEEAUE_CODES field of 323

CXIT control block *(continued)*
CEEAUE_FBCODE field of 328
CEEAUE_FLAGS field of
CEEAUE_DUMP field of 326
format of the 325
CEEAUE_FUNC field of 324
CEEAUE_LEN field of 324
CEEAUE_OPTION field of 327
CEEAUE_PARM field of 326
CEEAUE_USER field of
user word parameter of CEEBINT and 333
CEEAUE_WORK field of 327

D

DATA compile-time option, effect on storage 88, 89
Data Language/I (DL/I)
See DL/I DOS/VS
data type
guidelines for, when writing callable services 401
database rollback
assembler user exit and DB2 312
assembler user exit and DL/I 315
how CICS handles a 307
date and time
services
GETIME system macro and 363
summary 214
DB2
running applications with 311
specifying run-time options with 311
DEBUG run-time option
description 33
debug tool
CEEBINT and 332
CEEBXITA and 322
condition handling model and 114
debugging
ABPERC run-time option and 108
DELETE command
EXEC CICS command 359
DELETE service routine for preinitialization
components of 384
return/reason codes for 384
delete_entry
CEEPIPI(init_sub_dp) and) and 381
CEEPIPI(init_sub) and 381
DEPTHCONDLMT run-time option
description 33
DEQ 359
dereferencing 404
DETACH 359
DFHECI (EXEC CICS interface stub) 300
DFHELII (EXEC CICS interface stub) 300
DFSORT/VSE
condition handling for 410, 413
native invocations of 409, 410
user exits associated with 410
DISPLAY statement
default file for 202
OUTDD compile-time option for COBOL 203
DL/I DOS/VS
C considerations 313, 405
CEETDLI interface 313
coding a main routine to run with 53
condition handling under 314
list of DLI interfaces 313
OPTIONS(BYADDR) and 314
PLIST and EXECOPS interactions 406
PLIST run-time option and 53
specifying run-time options with 314
SYSTEM(DLI) compile-time option and 53, 56, 314, 407
TRAP run-time option 314
DOS/VS VM/SP Sort Merge Version 2
See DFSORT/VSE
DSA (dynamic save area)
See also stack, frame
register 13 and 336
dump
CEEBXITA assembler user exit and 322, 326
for CICS 303, 309
LE/VSE
PDUMP system macro and 363
dynamic call
C, under CICS 309
external references resolved at run time when made 7
VS COBOL II, under CICS 309
dynamic routines 5
dynamic save area (DSA)
See DSA (dynamic save area)

E

EDC facility ID 198
EDCYCROP program, to display CICS-wide run-time option settings 45
EIB (exec interface block)
calls within same HLL and 309
user-written condition handlers, EXEC CICS commands and 306
enablement
condition handling step
definition of exceptions 106
discussion of 107, 109
TRAP run-time option and 108
enclave
definition 77
HLLs and 77, 78
main routines and 77
management of LE/VSE resources 78
multiple 78
nested
created by Csystem() function 393, 395
created by EXEC CICS LINK or EXEC CICS XCTL 393, 394, 395
MSGFILE filenames and 200, 397
relationship with C main() functions 77
relationship with COBOL run units 77
relationship with processes 77
role in Language Environment program management model 80
enclave *(continued)*
subroutines and 77
termination
behavior 71
with abend 326
with assembler routine 336
with HLL user exit 332
end_seq
See preinitialization facility, CEEPIPI(end_seq)
ENQ 359
ENTRY linkage editor control statement 24, 25, 26
entry point
defaults for each language 6
defining, when link-editing a fetchable PL/I phase 8
link-edit map contains entry point names 26
ENV run-time option
C interface to DL/I 313
description 33
ENVAR run-time option
description 33
environment, common 3
equality, testing a condition token for 175
equivalence, testing a condition token for 174
ERRCOUNT run-time option
condition handling model and 110
description 33
error message (severity 2) 198
ESD map of defined and longnames 14
Euro support 249
examples
building a condition token, in C 158
CEE5CTY—set default country
with CEEFMDT and CEEDATM 249, 255
CEEDATE—convert Lilian date to character format
with CEEDAYS and CEEDYWK 238
CEEDATM—convert seconds to character format
with CEE5CTY and CEEFMDT 249, 255
with CEESECS callable service 226, 231
with CEESECS, CEESECI, and CEEISEC 231
CEEDAYS—convert date to Lilian format
with CEEDATE and CEEDYWK 238
CEEDCOD—decompose a condition token
with CEEMOUT, CEENCOD, CEEMGET, and CEEMSG 160, 164
CEEDYWK—calculate day of week from Lilian date
with CEEDATE and CEEDAYS 238

examples *(continued)*
CEEFMDT—get default date and time format
with CEE5CTY and CEEDATM 249, 255
CEEFMON—format monetary string 258, 260
CEEFTDS—format date and time into character string 261, 263
CEEHDLR—register user-written condition handler
calling from assembler 165, 170
with CEE5SPM, CEE5GRN, and CEEMOUT 164
with CEEGTST, CEECZST and CEEMRCR 149, 157
with CEESGL, CEEGQDT, and CEEMRCR 158
CEEISEC—convert integers to seconds
with CEESECS, CEESECI, and CEEDATM 231
CEELCNV—query locale numeric conventions
with CEESETL 264, 266
CEEMGET—get a message
with CEEMOUT, CEENCOD, CEEDCOD, and CEEMSG 160, 164
CEEMOUT—dispatch a message
with CEEHDLR, CEE5SPM, and CEE5GRN 164
with CEENCOD, CEEMGET, CEEDCOD, and CEEMSG 160, 164
CEEMSG—get, format, and dispatch a message
with CEEMOUT, CEENCOD, CEEMGET, and CEEDCOD 160, 164
CEENCOD—construct a condition token
with CEEMOUT, CEEMGET, CEEDCOD, and CEEMSG 160, 164
CEEQCEN—query century window 219
CEEQDTC—query locale, date, and time conventions
with CEESETL 267, 269
CEEQRYL—query active locale environment
with CEESETL 273, 275
with CEESTXF 276, 278
CEESCEN—set century window 219
CEESCOL—compare string collation weight 270, 272
CEESECI—convert seconds to integers
with CEESECS, CEEISEC, and CEEDATM 231
CEESECS—convert timestamp to number of seconds
multiple calls to 222, 225
using CEEDATM with 226, 231
with CEESECI, CEEISEC, and CEEDATM 231

examples *(continued)*
CEESETL—set locale operating environment
with CEELCNV 264, 266
with CEEQDTC 267, 269
CEESSLOG—calculate logarithm base e 290
CEESTXF—transform string into collation weights
with CEEQRYL 276, 278
link-editing a PL/I-fetchable phase 7
math services 289
querying and setting the century window 219
exceptions
historical definition 106
LE/VSE definition 106
EXEC CICS command
ABEND 306, 307, 326
DELETE 359
FREEMAIN 303, 363
GETMAIN 303, 363
HANDLE ABEND
assembler user exit and 307
CEEHDLR callable service and 359
CEEHDLU callable service and 359
CEESGL callable service and 359
table of equivalent LE/VSE services 359
TRAP run-time option and 306
user-written condition handlers and 306
HANDLE AID
assembler user exit and 307
user-written condition handlers and 306
HANDLE CONDITION
assembler user exit and 307
user-written condition handlers and 306
IGNORE CONDITION 306, 307
LINK
assembler routines and 363
behavior of nested enclaves created by 394
C and 309
DOS/VS COBOL and 310
program management model and 298
run-time options and 394
LOAD 359
POP HANDLE
assembler user exit and 307
user-written condition handlers and 306
PUSH HANDLE
assembler user exit and 307
user-written condition handlers and 306
RETURN 310
XCTL
assembler routines and 363
behavior of nested enclaves created by 394, 395
C and 309

EXEC CICS command *(continued)*
XCTL *(continued)*
DOS/VS COBOL and 310
program management model and 298
run-time options and 394
EXEC CICS interface stubs
See DFHECI, DFHELII
exec interface block (EIB)
See EIB (exec interface block)
EXEC job control statement
EXECOPS run-time option and 30
invoking linkage editor 25
syntax for executing an application 29
syntax for specifying run-time options 30, 39
EXECOPS run-time option
CEENTRY macro and 342
considerations when specifying run-time options 39
description 33
EXEC job control statement and 30
EXHIBIT for DOS/VS COBOL
default output file of 202
no support for, under CICS 202
external data
constructed reentrancy and 58
preinitialization and 366
scope of, in Language Environment program management model 77, 78

F

facility ID
each language component has a 198
part of condition token 172, 198
part of messages 198, 199
feedback code
condition manager and 173
condition token and 173, 175
guidelines for writing callable services and 401
in callable services 173
omitting 175
symbolic feedback code in condition handling 175, 181
FETCH statement
link-editing PL/I-fetchable phases 7
PL/I fetching PL/I 310
files used for link-editing 23
FREESTORE service routine for preinitialization
components of 385
return/reason codes for 386
FREEVIS system macro 363
See EXEC CICS command, FREEMAIN
freopen 201

G

genxlt utility 31
GET system macro 363
GETIME command
LE/VSE date/time services and 363

GETIME command (continued)
  table of equivalent LE/VSE services 363
GETSTORE service routine for preinitialization
  components of 384
  return/reason codes for 385
GETVIS storage required by LE/VSE 29
GETVIS system macro 363
  See EXEC CICS command, GETMAIN
global assembler user exit 319
global error table 121
  See condition handling
glossary 417

H

HANDLE ABEND EXEC CICS command
  assembler user exit and 307
  CEEHDLR and 359
  CEEHDLU and 359
  CEESGL and 359
  table of equivalent LE/VSE services 359
  TRAP run-time option and 306
  user-written condition handlers and 306
handle cursor
  definition 106
  promote action and 139
header files
  `stdlib.h` and the `__R1` and `__osplist` macros 404
  symbolic feedback code files and 176, 178
HEAP run-time option
  description 33
heap storage
  See also additional heap
  callable services for
    relationship to GETVIS/FREEVIS host services 359, 363
  examples of HLL data stored in 85
  heap element
    heap storage model and 85, 87, 101
  heap increment
    when allocated 85
  heap storage model 85, 87
  initial heap segment 85
    heap storage model and 87
    performance and 87
    when allocated 85
  lifetime of 85
  MODE considerations of 87
  program management model and 78
  reallocating (changing size of)
    See callable services, CEECZST—reallocate (change size of) storage
  RPTSTG run-time option and 87
  threads and 85
  tuning 87
heap_id 0
  See also heap storage, initial heap segment
  See initial heap
homepage, VSE xxiii

I

I/O
  See input/output
IBM facility ID 198
iconv utility 31
IGZ facility ID 198
IGZERRE 364
IJSYS workfiles used for link-editing 23
ILBDSET0 364
ILC (interlanguage communication)
  benefits of LE/VSE support 103
  link-editing ILC applications 6
  overlay programs and 337
INCLUDE file 185
INCLUDE linkage editor control statement 24, 25, 26
  application-specific assembler user exit and 320
informational message (severity 0) 198
init_main
  See preinitialization facility, CEEPIPI(init_main)
init_sub
  See preinitialization facility, CEEPIPI(init_sub)
init_sub_dp
  See preinitialization facility, CEEPIPI(init_sub_dp)
initial heap
  heap storage model and 87
initial heap segment
  See heap storage, initial heap segment
initial stack segment 83
  performance and 84
  stack storage model and 83
  when allocated 83
initializing 335
  See also preinitialization facility
  initialization routines 5
  nested enclave
    CEEBXITA's function code for 324
  using CEEBXITA assembler user exit for
    CEEBXITA behavior 320
    function code for 324
input/output
  CICS restrictions 298
  LE/VSE default message file attributes 199
insert data
  user-created
    assigning values to 196
installation-wide assembler user exit 319
interlanguage communication (ILC)
  See also external data
  See ILC (interlanguage communication)
interleaved output to MSGFILE 201, 202
Internet address, VSE homepage xxiii
interrupts
  See program interrupts
intrinsic functions 86
ISI (instance specific information)
  callable service feedback code and 173
  description 173
  q_data_token 182

L

L-names
  LIBRARY control statement and 16
  mapping to S-names 18
  RENAME control statement and 16
  UPCASE prelink option and 19
language environment
  introduction 3
LE/VSE
  introduction 3
  library structure 5
LIBDEF statement
  format in JCL 29
librarian automatic library lookup (AUTOLINK) function
  ACTION linkage editor control statement and 24, 25
  files required by the linkage editor and 23
libraries, linkage editor use of 23
library routine retention 337
LIBRARY statement
  prelinker and 16
library, LE/VSE
  See LE/VSE, library structure
LIBSTACK run-time option
  description 33
Lilian date
  calculate day of week from (CEEDYWK) 214
  convert date to (CEEDAYS) 214
  convert to character format (CEEDATE) 214
  return current local date as a (CEELOCT) 214
  return GMT as a (CEEGMT) 214
link-editing
  basics of linking and running 5
  CICS considerations 299
  detecting errors 28
  diagram of linkage editor processing 22
  example of 25
  files used for 23
  ILC applications 6
  input to the linkage editor 22
  options 24, 25
  PL/I-fetchable phases 7
  using INCLUDE statement to include additional modules as input 26
  VS COBOL II NORES considerations 7
  writing JCL for the linkage editor 25
linkage editor
  control statements 24, 25
  function 5
  generating a link-edit map 24, 25
  input to 22
  messages, where they go 23, 28
  writing JCL for 23

LOAD service routine for preinitialization
components of 383
return/reason codes for 383, 384
LOAD system macro 363
local
data 78
locale callable services 257
LONGNAME compile-time option 11

M

macro
__csplist 405
__osplist 404
__pcblist 405
__R1 404
CEECAA
relationship to CEEENTRY 341
syntax description 344
CEECIB 345
CEEDSA
relationship to CEEENTRY 341
syntax description 345
CEEENTRY
relationship to CEECAA 341
relationship to CEEDSA 342
relationship to CEEPPA 341, 342
relationship to CEETERM 341
syntax description 341
CEEFETCH 350
CEELOAD 348
CEEPPA
relationship to CEEENTRY 341
syntax description 345
CEERELES 353
CEETERM
relationship to CEEENTRY 341
syntax description 344
CEEXOPT
description 35
sample of CEECOPT modified using 42
sample of CEEDOPT modified using 41
sample of CEEUOPT modified using 43
usage notes for 44, 45
CEEXPIT 365
CEEXPITS 365
CEEXPITY 365
main routine
assembler main
example of a simple 355
example of main calling a sub 356
register values on entry to 335
determining 77
nested enclave considerations 393
position in Language Environment program management model 77
preinitialization of 363, 369
management of resources 78
management, program 75, 80
mapping
L-names to S-names 18
math services
about 287
math services *(continued)*
examples using 289
MERGE (COBOL verb)
condition handling considerations 410, 413
overview 409
user exit triggered by 410
message file
C stderr and stdout output and 201
CICS considerations 200
COBOL condition handling semantics and 129
COBOL DISPLAY statement and 203
LE/VSE's default destinations 199
nested enclave considerations 200, 397
PL/I I/O statements 203
specifying filename of 200
using CEEBLDTX to assemble 185
message handling
See also ISI (instance specific information)
CESE transient data queue and 308
relationship to fc parm of callable services 173, 175
specifying filename of message file 200
message module table 185
messages
complete listing of
See LE/VSE Debugging Guide and Run-Time Messages
condition token and 198, 199
directing to an I/O device 200
example 199
facility ID 198
message prefixes 199
redirecting stderr, stdout and printf() output 201
severity codes and values 198
using in your application 200
migration
See also IBM COBOL for VSE/ESA Migration Guide
See IBM PL/I for VSE/ESA Migration Guide
MODE linkage editor control statement 24, 25
models, architectural
program management 75, 80
MSGFILE run-time option 6, 129
COBOL condition handling semantics and 129
default destination 199
description 33
different treatment under CICS 303
specifying filenames across nested enclaves 200, 397
SYSPRINT file and 203
MSGQ run-time option
description 33
MSGRTN service routine for preinitialization
components of 386
return/reason codes for 386
multiple
enclaves 78
multiple *(continued)*
processes 77
threads 79

N

NAB (next available byte)
assembler main routine and 336
assembler subroutine and 336
CEEENTRY macro and 342
national language support (NLS)
message handling and 199
NATLANG run-time option
description 33
messages and 199
natural reentrancy 57
nested conditions 140
nested enclave
See also enclave, nested
See also EXEC CICS command, LINK
See also EXEC CICS command, RETURN
See EXEC CICS command, XCTL
NLS (national language support)
See national language support (NLS)
nonoverridable run-time options 36, 327

O

omitted parameter 175
condition manager reaction to 175
considerations when writing a callable service 401
ON EXCEPTION clause 128
ON SIZE ERROR clause 128, 132
OPEN system macro 363
OPTIONS(BYADDR)
description 55
DL/I considerations 314, 407
specifying with OPTIONS(BYVALUE) is an error 55
SYSTEM(CICS) and 407
when it is the default 56
OPTIONS(BYVALUE)
description 55
OPTIONS(NOEXECOPS) and 56
rules for specifying 55
specifying with OPTIONS(BYADDR) is an error 55
SYSTEM(CICS) and 407
when it is the default 56
osplist macro 404
OUTDD compile-time option for COBOL 203
overflow condition
C conditions and default system actions 122
C SIGFPE condition and 122
COBOL ON SIZE ERROR clause and 132
enabling and disabling 108
overlay programs 337
overridable/nonoverridable run-time options 36, 327

P

parallel processing 79
parameter
- *See also* omitted parameter
- list
  - accessing by using macros 404, 405
  - assembler 336
  - relationship to argument list 50
- list format
  - effect of EXECOPS run-time option on 405, 406
  - how interaction of EXECOPS and PLIST run-time options affects 405, 406
  - PLIST run-time option and 405
- list pointer 51
- passing
  - by reference 50, 51
  - by value 50, 51
  - C passing styles 403, 405
  - directly 50, 51
  - indirectly 50, 51
  - passing styles permitted by LE/VSE 51, 401

PARM parameter of JCL EXEC statement 25, 30, 36, 368
partition (CICS) 297
pcblist macro 405
PCT (Program Control Table) 304
PDUMP system macro
- CEE5DMP callable service and 363
- table of equivalent LE/VSE services 363

percolate action
- C condition handling and 124
- COBOL condition handling and 129
- compared to promote and resume actions 116
- condition handling model and 110
- user-written condition handler syntax for 139

PHASE linkage editor control statement 24, 25
PIPI table
- *See* preinitialization facility, PIPI table

PL/I
- BYADDR
  - functions 55
  - must be specified if SYSTEM(DLI) specified 407
- BYVALUE
  - functions 55
  - must be specified if SYSTEM(CICS) specified 407
- calls under CICS 310
- condition handling 132, 136
- examples
  - CEE5CTY, CEEFMDT, CEEDATM 254, 255
  - CEE5RPH, CEECRHP, CEEGTST, CEECZST, CEEFRST and CEEDSHP 100, 101
  - CEEDAYS, CEEDATE, CEEDYWK 243, 244
  - CEEFMON—format monetary string 260

PL/I *(continued)*
- examples *(continued)*
  - CEEFTDS—format date and time into character string 263
  - CEEGTST and CEEFRST 94, 96
  - CEELCNV and CEESETL 266
  - CEEMOUT, CEENCOD, CEEMGET, CEEDCOD, CEEMSG 210
  - CEEQCEN and CEESCEN 221
  - CEEQDTC and CEESETL 269
  - CEESCOL—compare string collation weight 272
  - CEESECS and CEEDATM 230, 231
  - CEESECS, CEESECI, CEEISEC, CEEDATM 236
  - CEESECS, multiple calls to 225
  - CEESETL and CEEQRYL 275
  - CEESSLOG 292
  - CEESTXF and CEEQRYL 278
  - coding main routines to receive inbound parm list 52, 55
- interfaces to DL/I from 313
- link-editing fetchable phases 7, 8
- linked list, building 94, 96
- MSGFILE run-time option and SYSPRINT 203
- NOEXECOPS option on Procedure statement 406
- parameter passing style 51
- PLIXOPT variable, specifying run-time options in 37, 300, 311, 314
- run-time options, specifying from 35, 45
- variables, where stored 85

PLIRETC subroutine
- CICS support for 299, 307

PLIRETV intrinsic function
- CICS support for 299

PLIST run-time option
- argument list format and 405, 406
- C interface to DL/I 313
- description 33

PLITDLI interface to DL/I 313
PLIXOPT variable, specifying run-time options in 37, 300, 311, 314
POST system macro 363
PPA (Program Prolog Area) 341, 345
PPT (Processing Program Table) 298
pragma
- `#pragma runopts`
  - affecting argument list format with 405, 406
  - DL/I and 313
  - specifying run-time options with 39, 300, 311, 314

PRD2.SCEEBASE and PRD2.SCEECICS sublibraries
- default installation sublibraries 5

preinitialization facility
- benefits of 363
- CEEPIPI(add_entry)
  - CEEPIPI(init_main) and 380
  - CEEPIPI(init_sub) and 380
  - function code for 364
  - return codes from 381

preinitialization facility *(continued)*
- CEEPIPI(add_entry) *(continued)*
  - syntax description 380
- CEEPIPI(call_main)
  - assembler user exits and 373
  - CEEPIPI(init_main) and 373
  - COBOL STOP RUN and 373
  - function code for 364
  - reentrancy considerations 366
  - return codes from 374
  - syntax description 373
- CEEPIPI(call_sub_addr) 371, 376, 377
  - function code for 364
  - return codes from 377
  - syntax description 376
- CEEPIPI(call_sub)
  - CEEPIPI(init_sub) and 373, 375
  - CEEPIPI(term) and 379
  - COBOL STOP RUN and 374, 375, 376
  - function code for 364
  - return codes from 375
  - syntax description 374
- CEEPIPI(delete_entry)
  - CEEPIPI(init_sub_dp) and 381
  - CEEPIPI(init_sub) and 381
  - function code for 381
  - return codes for 381
  - syntax description 381
- CEEPIPI(end_seq) 378
  - function code for 364
  - return codes from 378
  - syntax description 378
- CEEPIPI(init_main)
  - CEEPIPI(add_entry) and 380
  - CEEPIPI(call_main) and 373
  - CEEPIPI(term) 379
  - function code for 364
  - return codes from 370
  - specifying service routines in 369
  - syntax description 369
- CEEPIPI(init_sub_dp)
  - function code for 364
  - return codes from 372
  - syntax description 371
- CEEPIPI(init_sub)
  - CEEPIPI(add_entry) and 380
  - CEEPIPI(call_sub) and 373, 375
  - CEEPIPI(term) and 379
  - function code for 364
  - return codes from 371
  - specifying service routines in 371
  - syntax description 370
- CEEPIPI(start_seq)
  - function code for 364
  - return codes from 378
  - syntax description 377
- CEEPIPI(term)
  - CEEPIPI(call_sub) and 379
  - CEEPIPI(init_main) and 379
  - CEEPIPI(init_sub) and 379
  - function code for 364
  - return codes from 379
  - syntax description 379
- CEEXPIT macro 365
- CEEXPITS macro 365
- CEEXPITY macro 365

preinitialization facility *(continued)*
IGZERRE (COBOL interface to preinitialization) 364
ILBDSET0 (COBOL interface to preinitialization) 364
PIPI table
add entry to 380
CEEPIPI(call_main) and 373
CEEPIPI(call_sub_addr) and 376
CEEPIPI(call_sub) and 375, 376
CEEPIPI(end_seq) and 378
CEEPIPI(init_main) and 369
CEEPIPI(init_sub_dp) and 371
CEEPIPI(init_sub) and 370
CEEPIPI(start_seq) and 377
generate entry within 365
generate heading for 365
identify end of 365
introduction to 364
restrictions against nested routines in 381
service routines for 382
See also @DELETE, @LOAD, @FREESTORE, @STORE
allocating storage for 384
AMODE/RMODE requirements of 383
freeing storage of 385
in CEEPIPI(init_main) 369, 382
in CEEPIPI(init_sub) 371, 382
relationship to each other 383
vector format 382
prelinker
constructed reentrancy 57
functions 11
how it maps L-names to S-names 18
INCLUDE statement and 15
LIBRARY statement and 16
prelink options 19
prelinker map 13
RENAME statement and 16
when it has to be used 11
process
assembler user exit for termination of 324
current support for 77
definition 77
relationship to enclaves 77
role in Language Environment program management model 80
termination of assembler routines and 336
Processing Program Table (PPT) 298
program interrupts
abend codes and return codes 72
condition handling and 106, 108
SORT/MERGE and 410
under CICS 306, 308
under DL/I 314
under SORT/MERGE 410
user exits and 323
program management model
diagram of 80
terminology of 77, 79
Program Prolog Area (PPA) 341, 345
prolog 341, 345
promote action
compared to percolate and resume actions 116
condition handling model and 110
user-written condition handler syntax for 139
PRV (pseudoregister vector) listings 27
PUT system macro 363

Q

q_data structure 182

R

R1 macro 404
READ system macro 363
READY TRACE statement 202
reason code
CEEPIPI(call_main) and 374
CEEPIPI(call_sub) and 375
in user exits 323, 325
summary of LE/VSE codes 70
under CICS 307
recursion
allowed in user-written condition handlers 139
Language Environment program management model and 78
REDIR run-time option
description 33
redirections
of `stderr`, `stdout` and `printf()` output 201
reentrancy
advantages of 57
C routines and
constructed reentrancy 57
limitations of reentrancy 57
natural reentrancy 57
procedure for generating reentrant phases in 58
reentrant routines split into two parts 58
CEEPIPI(call_main) and 366
CEEPIPI(delete_entry) 382
CICS routines and 57
COBOL RENT compile-time option and 58
modified CEEBXITA must be reentrant 323
PL/I REENTRANT procedure option and 58
prelinker and 57
routines that must be reentrant 57
shared virtual area (SVA) and 58
RENAME control statement
how prelinkage utility maps L-names to S-names 18
syntax and usage notes 16
RENT compile-time option
effect on storage 88, 89
making C routines reentrant with 58
making COBOL routines reentrant with 58
RENT compile-time option *(continued)*
prelinker must be used when C source file compiled with 11
resident routines 5
resume action
definition 116
severity 2 or above IGZ conditions and 129
user-written condition handlers and 129, 138
resume cursor
definition 106
nested conditions and 140
return code
calculation 69
CEEAUE_RETURN field of CXIT control block and 324
CEEPIPI(call_main) and 374
CEEPIPI(call_sub) and 375
CEEPIPI(delete_entry) 381
in user exits 324
RETURN-CODE special register 69
RMODE
C considerations 9
for fetchable phases 8
for preinitialization facility 364
root segment in overlay program 337
routines
See main routine, subroutine
RPTOPTS run-time option 6
description 33
RPTOPTS(OFF) 45
RPTOPTS(ON) 45
RPTSTG run-time option 84
description 33
storage report generated by using to tune the stacks 84
RTEREUS run-time option
description 33
preinitialization and 364
run unit
for CICS 297
for COBOL
relationship to LE/VSE enclave 77
run-time environment, introduction 3
run-time options
ABPERC—exempt an abend from condition handling
See ABPERC run-time option
ABTERMENC—control abnormal enclave termination behavior
See ABTERMENC run-time option
AIXBLD—invoke AMS for COBOL
See AIXBLD run-time option
ALL31—indicates whether an application runs in AMODE(31)
See ALL31 run-time option
ANYHEAP—control unrestricted library heap storage
See ANYHEAP run-time option
application defaults for
See CEEUOPT options module
ARGPARSE—specify whether arguments are parsed
See ARGPARSE run-time option

run-time options *(continued)*
BELOWHEAP—control library heap storage below 16MB
See BELOWHEAP run-time option
CBLOPTS—specify format of COBOL argument
See CBLOPTS run-time option
CBLPSHPOP—control CICS commands
See CBLPSHPOP run-time option
CHECK—detect checking errors
See CHECK run-time option
COUNTRY—specify default date/time formats
See COUNTRY run-time option
DEBUG—activate COBOL batch debugging
See DEBUG run-time option
DEPTHCONDLMT—limit extent of nested conditions
See DEPTHCONDLMT run-time option
ENV—specify operating environment for C application
See ENV run-time option
ENVAR—set initial values for environment variables
See ENVAR run-time option
ERRCOUNT—specify number of errors allowed
See ERRCOUNT run-time option
EXECOPS—let run-time options be specified on command line
See EXECOPS run-time option
HEAP—control allocation of heaps
See HEAP run-time option
how nested enclaves get
enclaves created by C `system()` 395
enclaves created by EXEC CICS commands 394
IBM-supplied and installation defaults
See CEEDOPT options module
IBM-supplied and installation defaults for, under CICS
See CEECOPT options module
in the CEEPIPI interface to preinitialization 371, 373
in the user exit 322, 327
LIBSTACK—control library stack storage
See LIBSTACK run-time option
MSGFILE—specify filename of diagnostic file
See MSGFILE run-time option
MSGQ—specify number of ISI blocks allocated
See MSGQ run-time option
NATLANG—specify national language
See NATLANG run-time option
PLIST—specify format of C arguments
See PLIST run-time option
printing CICS-wide options to console 45

run-time options *(continued)*
REDIR—specify whether C standard input/output can be redirected
See REDIR run-time option
RPTOPTS—generate a report of run-time options
See RPTOPTS run-time option
RPTSTG—generate a report of storage used
See RPTSTG run-time option
RTEREUS—initialize a reusable COBOL environment
See RTEREUS run-time option
specifying 35, 45
See CICS, specifying run-time options for
See also running an application, specifying run-time options for
order of precedence 37
STACK—allocate stack storage
See STACK run-time option
STORAGE—control storage
See STORAGE run-time option
TERMTHDACT—specify type of information generated with unhandled error
See TERMTHDACT run-time option
TEST—indicate debug tool to gain control
See TEST run-time option
TRACE—activate LE/VSE run-time library tracing
See TRACE run-time option
TRAP—handle abends and program interrupts
See TRAP run-time option
UPSI—set UPSI switches
See UPSI run-time option
XUFLOW—specify program interrupt due to exponent underflow
See XUFLOW run-time option
run-time options, CICS-wide, printing 45
running an application
specifying run-time options for 30, 36
writing JCL to run an application 29

S

S-names
prelinker and
how L-names are mapped to S-names 18
save area 105
SCEEBASE and SCEECICS sublibraries
CEECOPT source file 36
CEEDOPT source file 35
default installation sublibraries 5
search order of sublibraries 29
service
routines
allocating storage for 384
AMODE/RMODE requirements of 383
freeing storage of 385

service *(continued)*
routines *(continued)*
in CEEPIPI(init_main) 369, 382
in CEEPIPI(init_sub) 371, 382
relationship to each other 383
vector format 382
service routines for preinitialization
See @DELETE, @FREESTORE, @GETSTORE, @LOAD, @MSGRTN
SETIME and SETT system macros 363
SETRP command
CEEHDLR callable service and 359
CEEHDLU callable service and 359
CEESGL callable service and 359
table of equivalent LE/VSE services 359
severe error message (severity 3) 198
severity
of conditions
CEEBXITA assembler user exit and 325
COBOL condition handling 129
condition token and 172
ERRCOUNT run-time option and 110
how to determine in a message 111, 198
TERMTHDACT run-time option and 113
unhandled conditions and 70, 111
shared virtual area (SVA)
See SVA (shared virtual area)
short-on-storage condition 303
SIGABRT
HLL user exit and 332
SIGTERM
HLL user exit and 332
slash (/)
specifying in parameter list 22, 38
SORT/MERGE
condition handling within 410, 413
overview of sort/merge operations 409, 410, 411
parameter list format under CICS 413
storage use 413
user exits triggered by 410
Sort/Merge II
See DFSORT/VSE
stack
frame 81, 83, 84
condition management model and 109
differentiated from Global Error Table model of condition handling 107, 121
HLL-specific condition handlers and 115
obtaining 105
stack frame zero 110, 114, 129
user-written condition handlers and 114
increment
stack storage model and 83
when allocated 83

stack *(continued)*
storage
Language Environment program management model and 79
LE/VSE stack storage model 84
LE/VSE storage management run-time options and GETVIS/FREEVIS 363
RPTSTG run-time option and 84
threads and 83
tuning 84
STACK run-time option 83, 84
description 33
LE/VSE stack storage model and 83
using with RPTSTG to tune the stack 84
standard streams 201
start_seq
See preinitialization facility, CEEPIPI(start_seq)
static data 337
STOP RUN
in a preinitialized environment 367
relationship to CEEPIPI(call_main) 373
STOP statement
for COBOL
CEEPIPI(call_sub) and 374, 375, 376
in a preinitialized environment 367
storage
GETVIS storage required 29
management model 81, 101
heap storage 85, 101
stack storage 83, 84
manager 81
operating system services for 359, 363
program storage required 29
service routines for 383
sort requirements 413
STORAGE run-time option
description 33
STXIT abnormal termination exit
sort and merge condition handling 410
STXIT system macro
CEEHDLR and 359
CEEHDLU and 359
CEESGL and 359
table of equivalent LE/VSE services 359
subroutine
assembler
examples using 356, 358
register values of 336
position in Language Environment program management model 77
preinitialization and 363
restriction regarding nested enclaves 393
success, testing a condition token for 174
SVA (shared virtual area)
reentrancy considerations 58
symbolic feedback code 175, 181
SYSIPT, linkage editor use of 23
SYSLNK, linkage editor use of 23
SYSLOG, linkage editor use of 23
SYSLST
default destinations of MSGFILE run-time option 199
destination when inserting messages in your application 202
linkage editor use of 23, 28
SYSPRINT as filename in MSGFILE run-time option 203
SYSRDR, linkage editor use of 23
system dump
See dump
SYSTEM PL/I compile-time option
SYSTEM(CICS) required when running under CICS 54, 56, 407
SYSTEM(DLI) required when running with DL/I DOS/VS 53, 56, 314, 407

T

termination 113
CEETERM macro and 343
enclave
as indicated in CEEAUE_ABND field of CEEAUE_FLAGS 326
as indicated in CEEAUE_ABTERM field of CEEAUE_FLAGS 325
CEEBXITA behavior during 322
CEEBXITA function codes for 324
terminating enclave created by an assembler routine 336
terminating enclave using CEEBINT HLL user exit 332
preinitialized routines and 363
process
CEEBXITA behavior during 322
CEEBXITA function code for 324
terminating process created by assembler routine 336
TERMTHDACT run-time option and 113
thread 112
termination imminent step
discussion of 111, 114
TERMTHDACT run-time option
condition message and 111
description 33
termination imminent step and 113
TEST run-time option
condition handling model and 114
description 33
maximum allowable length in CEEXOPT macro 44
thread
multiple 79
role in Language Environment program management model 80
stack storage and 83
token, condition
See condition token
TRACE run-time option
description 33
translator (CICS) 298, 306
TRAP run-time option
abends that occur in CEEBXITA and 323
ABPERC run-time option and 108
CEEBXITA assembler user exit and 308
CICS condition handling and 306
description 33
errors occurring in CEEBXITA and 323
how CEEAUE_ABND is affected by 326
nested enclaves and
enclaves created by C `system()` function 395
enclaves created by EXEC CICS LINK or EXEC CICS XCTL 394, 395
using with CEEAUE_CODES to exempt abends from condition handling 323
using with interfaces to DL/I 314

U

UPSI run-time option
description 33
user
exit
assembler 323
for initialization 321, 322, 366
for termination 322, 366
HLL 332
under CICS 324, 326, 327
under SORT/MERGE 410
heap (initial heap)
heap storage model and 87
return code
See also return code
C language constructs that generate 69
COBOL language constructs that generate 69
PL/I language constructs that generate 69
user-written condition handler
allowing nested conditions in 140
as opposed to condition manager 114
C `raise()` function and 122, 123
C `signal()` function and
terminology differences between C and LE/VSE 123
CEESGL callable service and 108
coding 137, 139
examples 141, 170
EXEC CICS commands that cannot be used with 306
in ILC applications 140
in nested condition handling 140
in SORT/MERGE condition handling 410
registering with CEEHDLR 115
role in LE/VSE condition management model 114
syntax for 138
TRAP run-time option and 108

USRHDLR run-time option
description 35, 137
using to register a user-written condition handler 139

V

VS COBOL II, considerations when link-editing NORES modules 7

W

WAIT and WAITM system macros 363
warning error message (severity 1) 198
working storage 78
writable static
handled by prelinker 11
writable static map 13
WRITE system macro 363
writeable static
See also external data
interface to
See CXIT control block
WTO system macro
CEEMOUT callable service and 363
table of equivalent LE/VSE services 363

X

XITPTR in interface to CEEBXITA 323
XUFLOW run-time option
description 33
using to manipulate the PSW 108

Readers' Comments — We'd Like to Hear from You

IBM Language Environment for VSE/ESA
Programming Guide
Version 1 Release 4 Modification Level 4

Publication No. SC33-6684-04

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
SC33-6684-04

Cut or Fold
Along Line

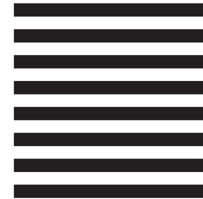
Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE



IBM Deutschland Entwicklung GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

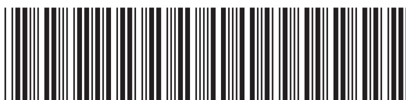
Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

File Number: S370-40
Program Number: 5686-CF7

Printed in USA

SC33-6684-04



Spine information:

LE/VSE

Programming Guide

Version 1
Release 4 Modification
Level 4

SC33-6684-04